DataStax Distribution of Apache Cassandra™ 3.x

Documentation

November 24, 2016

Contents

About Apache Cassandra

About this document

Welcome to the Cassandra documentation provided by DataStax. To ensure that you get the best experience in using this document, take a moment to look at the Tips for using DataStax documentation.

The landing pages provide information about supported platforms, product compatibility, planning and testing cluster deployments, recommended production settings, troubleshooting, third-party software, resources for additional information, administrator and developer topics, and earlier documentation.

Overview of Apache Cassandra

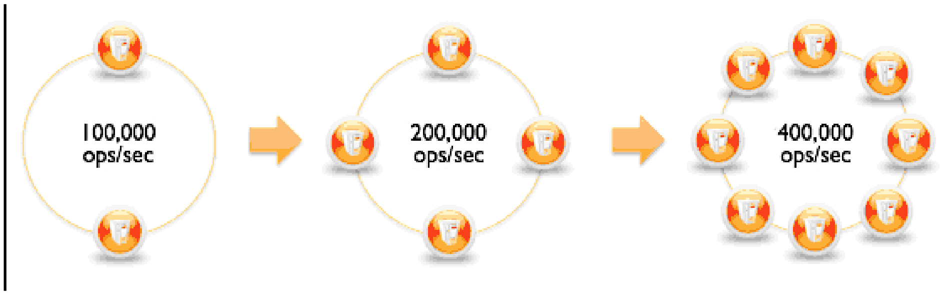
Apache Cassandra™ is a massively scalable open source NoSQL database. Cassandra is perfect for managing large amounts of structured, semi-structured, and unstructured data across multiple datacenters and the cloud. Cassandra delivers continuous availability, linear scalability, and operational simplicity across many commodity servers with no single point of failure, along with a powerful dynamic data model designed for maximum flexibility and fast response times.

The latest version of DataStax Distribution of Apache Cassandra 3.x is 3.9.

How does Cassandra work?

Cassandra's built-for-scale architecture means that it is capable of handling petabytes of information and thousands of concurrent users/operations per second.

Cassandra is a partitioned row store database	Cassandra's architecture allows any authorized user to connect to any node in any datacenter and access data using the CQL language. For ease of use, CQL uses a similar syntax to SQL. The most basic way to interact with Cassandra is using the CQL shell, cqlsh. Using `cqlsh`, you can create keyspaces and tables, insert and query tables, plus much more. This Cassandra release works with CQL for Cassandra 2.2 and later. If you prefer a graphical tool, you can use DataStax DevCenter. For production, DataStax supplies a number drivers so that CQL statements can be passed from client to cluster and back.
Automatic data distribution	Cassandra provides automatic data distribution across all nodes that participate in a *ring* or database cluster. There is nothing programmatic that a developer or administrator needs to do or code to distribute data across a cluster because data is transparently partitioned across all nodes in a cluster.
Built-in and customizable replication	Cassandra also provides built-in and customizable replication, which stores redundant copies of data across nodes that participate in a Cassandra ring. This means that if any node in a cluster goes down, one or more copies of that node's data is available on other machines in the cluster. Replication can be configured to work across one datacenter, many datacenters, and multiple cloud availability zones.
Cassandra supplies linear scalability	Cassandra supplies linear scalability, meaning that capacity may be easily added simply by adding new nodes online. For example, if 2 nodes can handle 100,000 transactions per second, 4 nodes will support 200,000 transactions/sec and 8 nodes will tackle 400,000 transactions/sec:



How is Cassandra different from relational databases?

Cassandra is designed from the ground up as a distributed database with peer-to-peer communication. As a best practice, queries should be one per table. Data is denormalized to make this possible. For this reason, the concept of JOINs between tables does not exist, although client-side joins can be used in applications.

What is NoSQL?

Most common translation is "Not only SQL", meaning a database that uses a method of storage different from a relational, or SQL, database. There are many different types of NoSQL databases, so a direct comparison of even the most used types is not useful. Database administrators today must be polyglot-friendly, meaning they must know how to work with many different RDBMS and NoSQL databases.

What is CQL?

Cassandra Query Language (CQL) is the primary interface into the Cassandra DBMS. Using CQL is similar to using SQL (Structured Query Language). CQL and SQL share the same abstract idea of a table constructed of columns and rows. The main difference from SQL is that Cassandra does not support joins or subqueries. Instead, Cassandra emphasizes denormalization through CQL features like collections and clustering specified at the schema level.

CQL is the recommended way to interact with Cassandra. Performance and the simplicity of reading and using CQL is an advantage of modern Cassandra over older Cassandra APIs.

The CQL documentation contains a data modeling topic, examples, and command reference.

How do I interact with Cassandra?

The most basic way to interact with Cassandra is using the CQL shell, `cqlsh`. Using cqlsh, you can create keyspaces and tables, insert and query tables, plus much more. If you prefer a graphical tool, you can use DevCenter. For production, DataStax supplies a number of drivers in various programming languages, so that CQL statements can be passed from client to cluster and back.

How can I move data to/from Cassandra?

Data is inserted using the CQL INSERT command, the CQL COPY command and CSV files, or sstableloader. But in reality, you need to consider how your client application will query the tables, and do data modeling first. The paradigm shift between relational and NoSQL means that a straight move of data from an RDBMS database to Cassandra will be doomed to failure.

What other tools come with Cassandra?

Cassandra automatically installs nodetool, a useful command-line management tool for Cassandra. A tool for load-stressing and basic benchmarking, cassandra-stress, is also installed by default.

What kind of hardware/cloud environment do I need to run Cassandra?

Cassandra is designed to run on commodity hardware with common specifications. In the cloud, Cassandra is adapted for most common offerings.

What's new in DataStax Distribution of Apache Cassandra 3.x

Note: Cassandra is now releasing on a tick-tock schedule.

The latest version of DataStax Distribution of Apache Cassandra 3.x is 3.9.

The CHANGES.txt describes the changes in detail. You can view all version changes by branch or tag in the drop-down list on the changes page.

New features Cassandra 3.2 and later

-graph option for cassandra-stress	`cassandra-stress` results can be automatically graphed for data visualization.
TTL for COPY FROM	A TTL value can be specified when copying from CSV files.
bulkloader can use third party authentication	The bulkloader has an option `-ap` for third-party authentication.
CREATE TABLE WITH ID	If a table is accidentally dropped, it can be recreated with its ID and the commitlog replayed to regain data.
Static columns can be indexed	In Cassandra 3.4 and later, static columns can be indexed.
New option for nodetool compact	In Cassandra 3.4 and later, addition of --user-definedcompact to `nodetool compact` to allow user to submit a list of files. Handy for dealing with low disk space or tombstone purging.
Display timestamp in sub-second precision	In Cassandra 3.4 and later, timestamp defaults to include sub-second precision.
nodetool gettimeout and nodetool settimeout	In Cassandra 3.4 and later, two nodetool commands to print out or set the value of a timeout in milliseconds.
jvm.options file for GC and some JVM options	Some JVM options have been moved from the `cassandra-env.sh` file into the new jvm.options file.
JBOD improvements	Improvements to SSTable partitioning by token range have improved JBOD compaction and backup. See Improving JBOD for more details. A new command is available to support the improvements, nodetool relocatesstables.
Clustering columns can be used in WHERE clause without secondary index	In Cassandra 3.6 and later, clustering columns without a secondary index can be used in a WHERE clause, provided the ALLOW FILTERING clause is also used.

Update and delete individual subfields of a user-defined type (UDT)	In Cassandra 3.6 and later, if a UDT has only non-collection fields, an individual field value can be updated or deleted.
PER PARTITION LIMIT	In Cassandra 3.6 and later, a query can be limited to return results from each partition, such as a "Top 3" listing.
CAS statistics added to nodetool proxyhistograms	In Cassandra 3.6 and later, CAS read and write latency is displayed for compare-and-set operations.
--hex-format option added to nodetool getsstables	In Cassandra 3.6 and later, an option to use a hex-formatted key to get SSTables is added to nodetool getsstables.
Static columns can now be used with SASI indexes	In Cassandra 3.6 and later, static columns can be used with SASI indexes.

New features released in Cassandra 3.0

Storage engine refactored	The Storage Engine has been refactored.
Materialized Views	Materialized views handle automated server-side denormalization, with consistency between base and view data.
Support for Windows	Support for Windows 7, Windows 8, Windows Server 2008, and Windows Server 2012. See DataStax Cassandra 3.0 Windows Documentation.
Operations improvements	
Addition of MAX_WINDOW_SIZE_SECONDS to DTCS compaction settings	Allow DTCS compaction governance based on maximum window size rather than SSTable age.
File-based Hint Storage and Improved Replay	Hints are now stored in files and replay is improved.
Default garbage collector is changed to G1	Default garbage collector is changed from Concurrent-Mark-Sweep (CMS) to G1. G1 performance is better for nodes with heap size of 4GB or greater.
Changed syntax for CREATE TABLE compression options	Made the compression options more consistent for `CREATE TABLE`.
Add nodetool command to force blocking batchlog replay	BatchlogManager can force batchlog replay using nodetool.
Nodetool over SSL	Nodetool can connect using SSL like cqlsh.

New nodetool options for hinted handoffs	Nodetool options `disablehintsfordc` and `enablehintsfordc` added. to selectively disable or enable hinted handoffs for a datacenter.
nodetool stop	Nodetool option added to stop compactions.
Other notable changes	
Requires Java 8	Java 8 is now required.
nodetool cfstats and nodetool cfhistograms renamed	Renamed `nodetool cfstats` to `nodetool tablestats`. Renamed `nodetool cfhistograms` to `nodetool tablehistograms`.
Native protocol v1 and v2 are dropped	Native protocol v1 and v2 are dropped in Cassandra 3.0.
DataStax AMI does not install Cassandra 3.0 or 3.x	You can install Cassandra 2.1 and earlier versions on Amazon EC2 using the DataStax AMI (Amazon Machine Image) as described in the AMI documentation for Cassandra 2.1. To install Cassandra 3.0 and later on Amazon EC2, use a trusted AMI for your platform and the appropriate install method for that platform.

Understanding the architecture

Architecture in brief

Cassandra is designed to handle big data workloads across multiple nodes with no single point of failure. Its architecture is based on the understanding that system and hardware failures can and do occur. Cassandra addresses the problem of failures by employing a peer-to-peer distributed system across homogeneous nodes where data is distributed among all nodes in the cluster. Each node frequently exchanges state information about itself and other nodes across the cluster using peer-to-peer gossip communication protocol. A sequentially written commit log on each node captures write activity to ensure data durability. Data is then indexed and written to an in-memory structure, called a memtable, which resembles a write-back cache. Each time the memory structure is full, the data is written to disk in an SSTables data file. All writes are automatically partitioned and replicated throughout the cluster. Cassandra periodically consolidates SSTables using a process called compaction, discarding obsolete data marked for deletion with a tombstone. To ensure all data across the cluster stays consistent, various repair mechanisms are employed.

Cassandra is a partitioned row store database, where rows are organized into tables with a required primary key. Cassandra's architecture allows any authorized user to connect to any node in any datacenter and access data using the CQL language. For ease of use, CQL uses a similar syntax to SQL and works with table data. Developers can access CQL through cqlsh, DevCenter, and via drivers for application languages. Typically, a cluster has one keyspace per application composed of many different tables.

Client read or write requests can be sent to any node in the cluster. When a client connects to a node with a request, that node serves as the coordinator for that particular client operation. The coordinator acts as a proxy between the client application and the nodes that own the data being requested. The coordinator determines which nodes in the ring should get the request based on how the cluster is configured.

Key structures

- Node

  Where you store your data. It is the basic infrastructure component of Cassandra.
- datacenter

  A collection of related nodes. A datacenter can be a physical datacenter or virtual datacenter. Different workloads should use separate datacenters, either physical or virtual. Replication is set by datacenter. Using separate datacenters prevents Cassandra transactions from being impacted by other workloads and keeps requests close to each other for lower latency. Depending on the replication factor, data can be written to multiple datacenters. datacenters must never span physical locations.
- Cluster

  A cluster contains one or more datacenters. It can span physical locations.
- Commit log

  All data is written first to the commit log for durability. After all its data has been flushed to SSTables, it can be archived, deleted, or recycled.
- SSTable

  A sorted string table (SSTable) is an immutable data file to which Cassandra writes memtables periodically. SSTables are append only and stored on disk sequentially and maintained for each Cassandra table.
- CQL Table

  A collection of ordered columns fetched by table row. A table consists of columns and has a primary key.

Key components for configuring Cassandra

- Gossip

  A peer-to-peer communication protocol to discover and share location and state information about the other nodes in a Cassandra cluster. Gossip information is also persisted locally by each node to use immediately when a node restarts.
- Partitioner

  A partitioner determines which node will receive the first replica of a piece of data, and how to distribute other replicas across other nodes in the cluster. Each row of data is uniquely identified by a primary key, which may be the same as its partition key, but which may also include other clustering columns. A partitioner is a hash function that derives a token from the primary key of a row. The partitioner uses the token value to determine which nodes in the cluster receive the replicas of that row. The Murmur3Partitioner is the default partitioning strategy for new Cassandra clusters and the right choice for new clusters in almost all cases.

  You must set the partitioner and assign the node a num_tokens value for each node. The number of tokens you assign depends on the hardware capabilities of the system. If not using virtual nodes (vnodes), use the initial_token setting instead.
- Replication factor

  The total number of replicas across the cluster. A replication factor of 1 means that there is only one copy of each row on one node. A replication factor of 2 means two copies of each row, where each copy is on a different node. All replicas are equally important; there is no primary or master replica. You define the replication factor for each datacenter. Generally you should set the replication strategy greater than one, but no more than the number of nodes in the cluster.
- Replica placement strategy

  Cassandra stores copies (replicas) of data on multiple nodes to ensure reliability and fault tolerance. A replication strategy determines which nodes to place replicas on. The first replica of data is simply the first copy; it is not unique in any sense. The NetworkTopologyStrategy is highly recommended for most deployments because it is much easier to expand to multiple datacenters when required by future expansion.

When creating a keyspace, you must define the replica placement strategy and the number of replicas you want.

- Snitch

  A snitch defines groups of machines into datacenters and racks (the topology) that the replication strategy uses to place replicas.

  You must configure a snitch when you create a cluster. All snitches use a dynamic snitch layer, which monitors performance and chooses the best replica for reading. It is enabled by default and recommended for use in most deployments. Configure dynamic snitch thresholds for each node in the cassandra.yaml configuration file.

  The default SimpleSnitch does not recognize datacenter or rack information. Use it for single-datacenter deployments or single-zone in public clouds. The GossipingPropertyFileSnitch is recommended for production. It defines a node's datacenter and rack and uses gossip for propagating this information to other nodes.
- The cassandra.yaml configuration file

  The main configuration file for setting the initialization properties for a cluster, caching parameters for tables, properties for tuning and resource utilization, timeout settings, client connections, backups, and security.

  By default, a node is configured to store the data it manages in a directory set in the cassandra.yaml file.

  In a production cluster deployment, you can change the commitlog-directory to a different disk drive from the data_file_directories.
- System keyspace table properties

  You set storage configuration attributes on a per-keyspace or per-table basis programmatically or using a client application, such as CQL.

Related reference

cassandra.yaml configuration file on page 73

Related information

Install locations on page 71

Internode communications (gossip)

Gossip is a peer-to-peer communication protocol in which nodes periodically exchange state information about themselves and about other nodes they know about. The gossip process runs every second and exchanges state messages with up to three other nodes in the cluster. The nodes exchange information about themselves and about the other nodes that they have gossiped about, so all nodes quickly learn about all other nodes in the cluster. A gossip message has a version associated with it, so that during a gossip exchange, older information is overwritten with the most current state for a particular node.

To prevent problems in gossip communications, use the same list of seed nodes for all nodes in a cluster. This is most critical the first time a node starts up. By default, a node remembers other nodes it has gossiped with between subsequent restarts. The seed node designation has no purpose other than bootstrapping the gossip process for new nodes joining the cluster. Seed nodes are *not* a single point of failure, nor do they have any other special purpose in cluster operations beyond the bootstrapping of nodes.

Attention: In multiple data-center clusters, include at least one node from each datacenter (replication group) in the seed list. Designating more than a single seed node per datacenter is recommended for fault tolerance. Otherwise, gossip has to communicate with another datacenter when bootstrapping a node.

Making every node a seed node is **not** recommended because of increased maintenance and reduced gossip performance. Gossip optimization is not critical, but it is recommended to use a small seed list (approximately three nodes per datacenter).

Failure detection and recovery

Failure detection is a method for locally determining from gossip state and history if another node in the system is down or has come back up. Cassandra uses this information to avoid routing client requests to unreachable nodes whenever possible. (Cassandra can also avoid routing requests to nodes that are alive, but performing poorly, through the dynamic snitch.)

The gossip process tracks state from other nodes both directly (nodes gossiping directly to it) and indirectly (nodes communicated about secondhand, third-hand, and so on). Rather than have a fixed threshold for marking failing nodes, Cassandra uses an accrual detection mechanism to calculate a per-node threshold that takes into account network performance, workload, and historical conditions. During gossip exchanges, every node maintains a sliding window of inter-arrival times of gossip messages from other nodes in the cluster. Configuring the phi_convict_threshold property adjusts the sensitivity of the failure detector. Lower values increase the likelihood that an unresponsive node will be marked as down, while higher values decrease the likelihood that transient failures causing node failure. Use the default value for most situations, but increase it to 10 or 12 for Amazon EC2 (due to frequently encountered network congestion). In unstable network environments (such as EC2 at times), raising the value to 10 or 12 helps prevent false failures. Values higher than 12 and lower than 5 are not recommended.

Node failures can result from various causes such as hardware failures and network outages. Node outages are often transient but can last for extended periods. Because a node outage rarely signifies a permanent departure from the cluster it does not automatically result in permanent removal of the node from the ring. Other nodes will periodically try to re-establish contact with failed nodes to see if they are back up. To permanently change a node's membership in a cluster, administrators must explicitly add or remove nodes from a Cassandra cluster using the nodetool utility.

When a node comes back online after an outage, it may have missed writes for the replica data it maintains. Repair mechanisms exist to recover missed data, such as hinted handoffs and manual repair with nodetool repair. The length of the outage will determine which repair mechanism is used to make the data consistent.

Data distribution and replication

In Cassandra, data distribution and replication go together. Data is organized by table and identified by a primary key, which determines which node the data is stored on. Replicas are copies of rows. When data is first written, it is also referred to as a replica.

Factors influencing replication include:

- Virtual nodes: assigns data ownership to physical machines.
- Partitioner: partitions the data across the cluster.
- Replication strategy: determines the replicas for each row of data.
- Snitch: defines the topology information that the replication strategy uses to place replicas.

Consistent hashing

Consistent hashing allows distribution of data across a cluster to minimize reorganization when nodes are added or removed. Consistent hashing partitions data based on the partition key. (For an explanation of partition keys and primary keys, see the Data modeling example in *CQL for Cassandra 2.2 and later*.)

For example, if you have the following data:

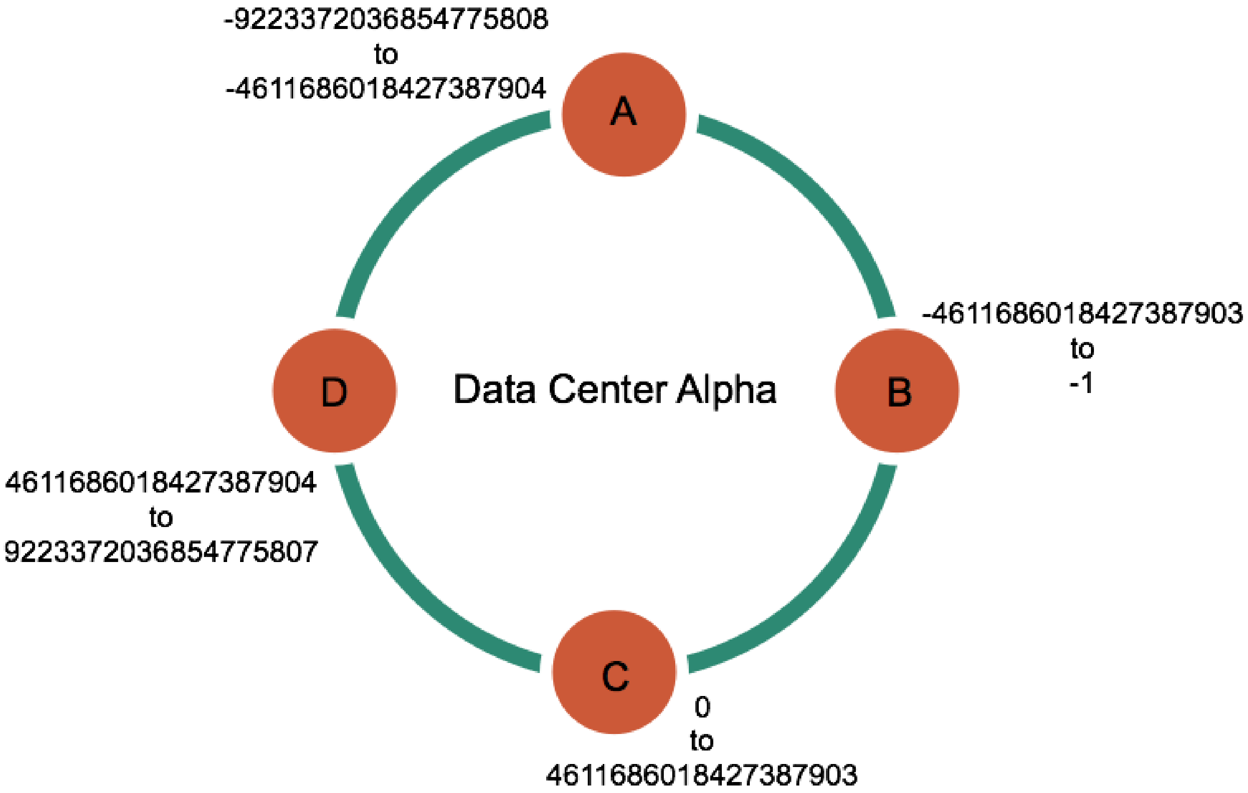
name	age	car	gender
jim	36	camaro	M
carol	37	bmw	F
johnny	12		M
suzy	10		F

Cassandra assigns a hash value to each partition key:

Partition key	Murmur3 hash value
jim	-2245462676723223822
carol	7723358927203680754
johnny	-6723372854036780875
suzy	1168604627387940318

Each node in the cluster is responsible for a range of data based on the hash value.

Figure: Hash values in a four node cluster



Cassandra places the data on each node according to the value of the partition key and the range that the node is responsible for. For example, in a four node cluster, the data in this example is distributed as follows:

Node	Start range	End range	Partition key	Hash value
A	-9223372036854775808	-4611686018427387904	johnny	-6723372854036780875

Node	Start range	End range	Partition key	Hash value
B	-4611686018427387903	-1	jim	-2245462676723223822
C	0	4611686018427387903	suzy	1168604627387940318
D	4611686018427387904	9223372036854775807	carol	7723358927203680754

Virtual nodes

Virtual nodes, known as Vnodes, distribute data across nodes at a finer granularity than can be easily achieved if calculated tokens are used. Vnodes simplify many tasks in Cassandra:

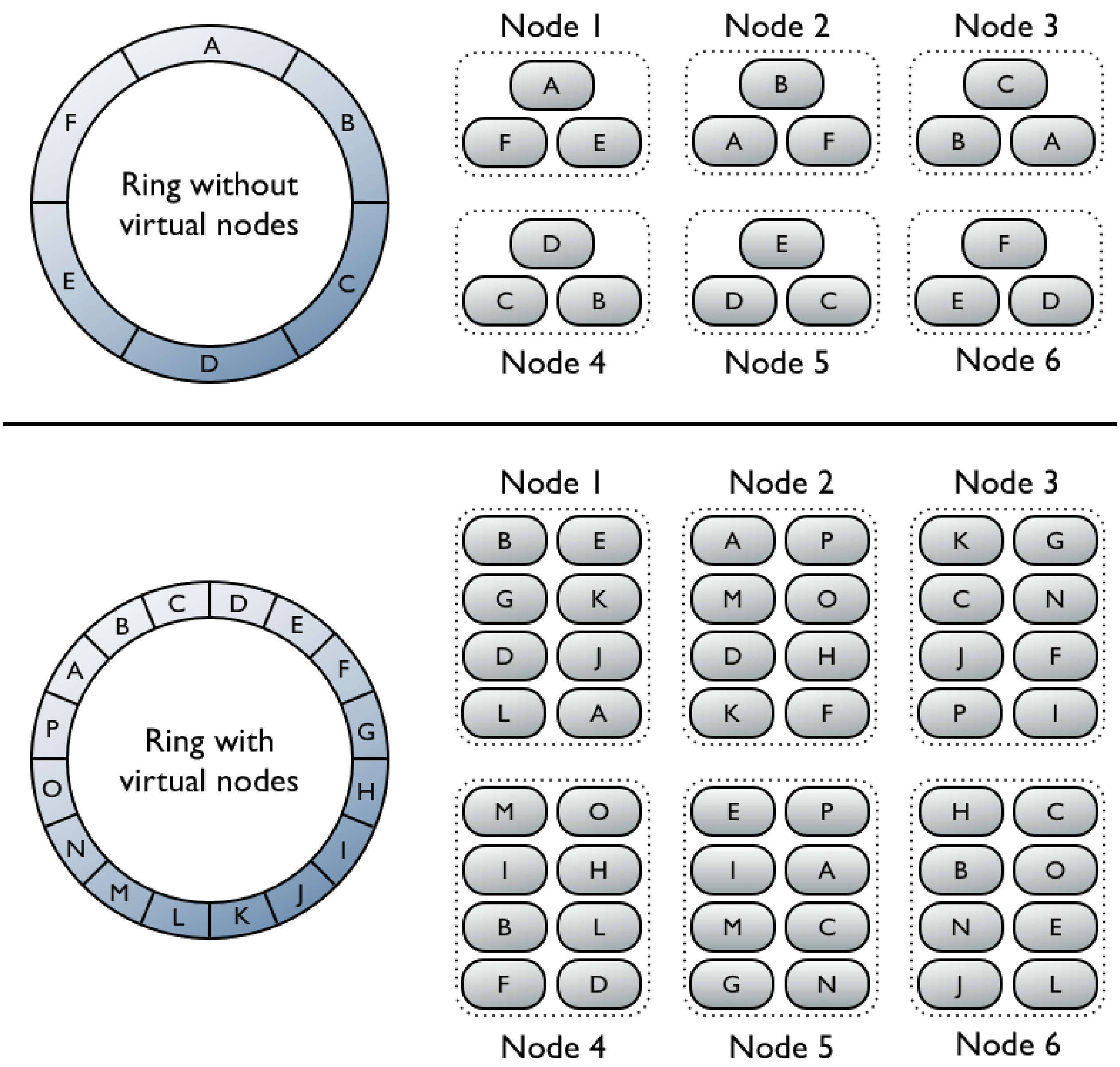
- Tokens are automatically calculated and assigned to each node.
- Rebalancing a cluster is automatically accomplished when adding or removing nodes. When a node joins the cluster, it assumes responsibility for an even portion of data from the other nodes in the cluster. If a node fails, the load is spread evenly across other nodes in the cluster.
- Rebuilding a dead node is faster because it involves every other node in the cluster.
- The proportion of vnodes assigned to each machine in a cluster can be assigned, so smaller and larger computers can be used in building a cluster.

For more information, see the article Virtual nodes in Cassandra 1.2. To convert an existing cluster to vnodes, see Enabling virtual nodes on an existing production cluster on page 124.

How data is distributed across a cluster (using virtual nodes)

Prior to Cassandra 1.2, you had to calculate and assign a single token to each node in a cluster. Each token determined the node's position in the ring and its portion of data according to its hash value. In Cassandra 1.2 and later, each node is allowed many tokens. The new paradigm is called virtual nodes (vnodes). Vnodes allow each node to own a large number of small partition ranges distributed throughout the cluster. Vnodes also use consistent hashing to distribute data but using them doesn't require token generation and assignment.

Figure: Virtual vs single-token architecture



The top portion of the graphic shows a cluster without vnodes. In this paradigm, each node is assigned a single token that represents a location in the ring. Each node stores data determined by mapping the partition key to a token value within a range from the previous node to its assigned value. Each node also contains copies of each row from other nodes in the cluster. For example, if the replication factor is 3, range E replicates to nodes 5, 6, and 1. Notice that a node owns exactly one contiguous partition range in the ring space.

The bottom portion of the graphic shows a ring with vnodes. Within a cluster, virtual nodes are randomly selected and non-contiguous. The placement of a row is determined by the hash of the partition key within many smaller partition ranges belonging to each node.

Data replication

Cassandra stores replicas on multiple nodes to ensure reliability and fault tolerance. A replication strategy determines the nodes where replicas are placed. The total number of replicas across the cluster is referred to as the replication factor. A replication factor of 1 means that there is only one copy of each row in the cluster. If the node containing the row goes down, the row cannot be retrieved. A replication factor of 2 means two copies of each row, where each copy is on a different node. All replicas are equally important;

there is no primary or master replica. As a general rule, the replication factor should not exceed the number of nodes in the cluster. However, you can increase the replication factor and then add the desired number of nodes later.

Two replication strategies are available:

- SimpleStrategy: Use only for a single datacenter and one rack. If you ever intend more than one datacenter, use the NetworkTopologyStrategy.
- NetworkTopologyStrategy: Highly recommended for most deployments because it is much easier to expand to multiple datacenters when required by future expansion.

SimpleStrategy

Use only for a single datacenter and one rack. SimpleStrategy places the first replica on a node determined by the partitioner. Additional replicas are placed on the next nodes clockwise in the ring without considering topology (rack or datacenter location).

NetworkTopologyStrategy

Use NetworkTopologyStrategy when you have (or plan to have) your cluster deployed across multiple datacenters. This strategy specifies how many replicas you want in each datacenter.

NetworkTopologyStrategy places replicas in the same datacenter by walking the ring clockwise until reaching the first node in another rack. NetworkTopologyStrategy attempts to place replicas on distinct racks because nodes in the same rack (or similar physical grouping) often fail at the same time due to power, cooling, or network issues.

When deciding how many replicas to configure in each datacenter, the two primary considerations are (1) being able to satisfy reads locally, without incurring cross data-center latency, and (2) failure scenarios. The two most common ways to configure multiple datacenter clusters are:

- Two replicas in each datacenter: This configuration tolerates the failure of a single node per replication group and still allows local reads at a consistency level of ONE.
- Three replicas in each datacenter: This configuration tolerates either the failure of one node per replication group at a strong consistency level of LOCAL_QUORUM or multiple node failures per datacenter using consistency level ONE.

Asymmetrical replication groupings are also possible. For example, you can have three replicas in one datacenter to serve real-time application requests and use a single replica elsewhere for running analytics.

Replication strategy is defined per keyspace, and is set during keyspace creation. To set up a keyspace, see creating a keyspace.

For more about replication strategy options, see Changing keyspace replication strategy on page 148.

Partitioners

A partitioner determines how data is distributed across the nodes in the cluster (including replicas). Basically, a partitioner is a function for deriving a token representing a row from its partition key, typically by hashing. Each row of data is then distributed across the cluster by the value of the token.

Both the Murmur3Partitioner and RandomPartitioner use tokens to help assign equal portions of data to each node and evenly distribute data from all the tables throughout the ring or other grouping, such as a keyspace. This is true even if the tables use different partition keys, such as usernames or timestamps. Moreover, the read and write requests to the cluster are also evenly distributed and load balancing is simplified because each part of the hash range receives an equal number of rows on average. For more detailed information, see Consistent hashing on page 14.

The main difference between the two partitioners is how each generates the token hash values. The RandomPartitioner uses a cryptographic hash that takes longer to generate than the Murmur3Partitioner. Cassandra doesn't really need a cryptographic hash, so using the Murmur3Partitioner results in a 3-5 times improvement in performance.

Cassandra offers the following partitioners that can be set in the cassandra.yaml file.

- `Murmur3Partitioner` (default): uniformly distributes data across the cluster based on `MurmurHash` hash values.
- `RandomPartitioner`: uniformly distributes data across the cluster based on `MD5` hash values.
- `ByteOrderedPartitioner`: keeps an ordered distribution of data lexically by key bytes

The `Murmur3Partitioner` is the default partitioning strategy for Cassandra 1.2 and later new clusters and the right choice for new clusters in almost all cases. However, the partitioners are not compatible and data partitioned with one partitioner cannot be easily converted to the other partitioner.

Note: If using virtual nodes (vnodes), you do **not** need to calculate the tokens. If not using vnodes, you **must** calculate the tokens to assign to the initial_token parameter in the cassandra.yaml file. See Generating tokens on page 130 and use the method for the type of partitioner you are using.

Related information

Install locations on page 71

Murmur3Partitioner

The Murmur3Partitioner is the default partitioner. The Murmur3Partitioner provides faster hashing and improved performance than the RandomPartitioner. The `Murmur3Partitioner` can be used with vnodes. However, if you don't use vnodes, you must calculate the tokens, as described in Generating tokens.

Use `Murmur3Partitioner` for new clusters; you cannot change the partitioner in existing clusters that use a different partitioner. The `Murmur3Partitioner` uses the `MurmurHash` function. This hashing function creates a 64-bit hash value of the partition key. The possible range of hash values is from -2^{63} to $+2^{63}-1$.

When using the `Murmur3Partitioner`, you can page through all rows using the token function in a CQL query.

RandomPartitioner

The `RandomPartitioner` was the default partitioner prior to Cassandra 1.2. It is included for backwards compatibility. The RandomPartitioner can be used with virtual nodes (vnodes). However, if you don't use vnodes, you must calculate the tokens, as described in Generating tokens.The `RandomPartitioner` distributes data evenly across the nodes using an MD5 hash value of the row key. The possible range of hash values is from 0 to 2^{127} -1.

When using the `RandomPartitioner`, you can page through all rows using the token function in a CQL query.

ByteOrderedPartitioner

Cassandra provides the `ByteOrderedPartitioner` for ordered partitioning. It is included for backwards compatibility. This partitioner orders rows lexically by key bytes. You calculate tokens by looking at the actual values of your partition key data and using a hexadecimal representation of the leading character(s) in a key. For example, if you wanted to partition rows alphabetically, you could assign an A token using its hexadecimal representation of 41.

Using the ordered partitioner allows ordered scans by primary key. This means you can scan rows as though you were moving a cursor through a traditional index. For example, if your application has user names as the partition key, you can scan rows for users whose names fall between Jake and Joe. This type of query is not possible using randomly partitioned partition keys because the keys are stored in the order of their `MD5` hash (not sequentially).

Although having the capability to do range scans on rows sounds like a desirable feature of ordered partitioners, there are ways to achieve the same functionality using table indexes.

Using an ordered partitioner is not recommended for the following reasons:

Difficult load balancing

More administrative overhead is required to load balance the cluster. An ordered partitioner requires administrators to manually calculate partition ranges based on their estimates of the partition key distribution. In practice, this requires actively moving node tokens around to accommodate the actual distribution of data once it is loaded.

Sequential writes can cause hot spots

If your application tends to write or update a sequential block of rows at a time, then the writes are not be distributed across the cluster; they all go to one node. This is frequently a problem for applications dealing with timestamped data.

Uneven load balancing for multiple tables

If your application has multiple tables, chances are that those tables have different row keys and different distributions of data. An ordered partitioner that is balanced for one table may cause hot spots and uneven distribution for another table in the same cluster.

Snitches

A snitch determines which datacenters and racks nodes belong to. They inform Cassandra about the network topology so that requests are routed efficiently and allows Cassandra to distribute replicas by grouping machines into datacenters and racks. Specifically, the replication strategy places the replicas based on the information provided by the new snitch. All nodes must return to the same rack and datacenter. Cassandra does its best not to have more than one replica on the same rack (which is not necessarily a physical location).

Note: If you change snitches, you may need to perform additional steps because the snitch affects where replicas are placed. See Switching snitches on page 147.

Dynamic snitching

By default, all snitches also use a dynamic snitch layer that monitors read latency and, when possible, routes requests away from poorly-performing nodes. The dynamic snitch is enabled by default and is recommended for use in most deployments. For information on how this works, see Dynamic snitching in Cassandra: past, present, and future. Configure dynamic snitch thresholds for each node in the cassandra.yaml configuration file.

For more information, see the properties listed under Failure detection and recovery on page 14.

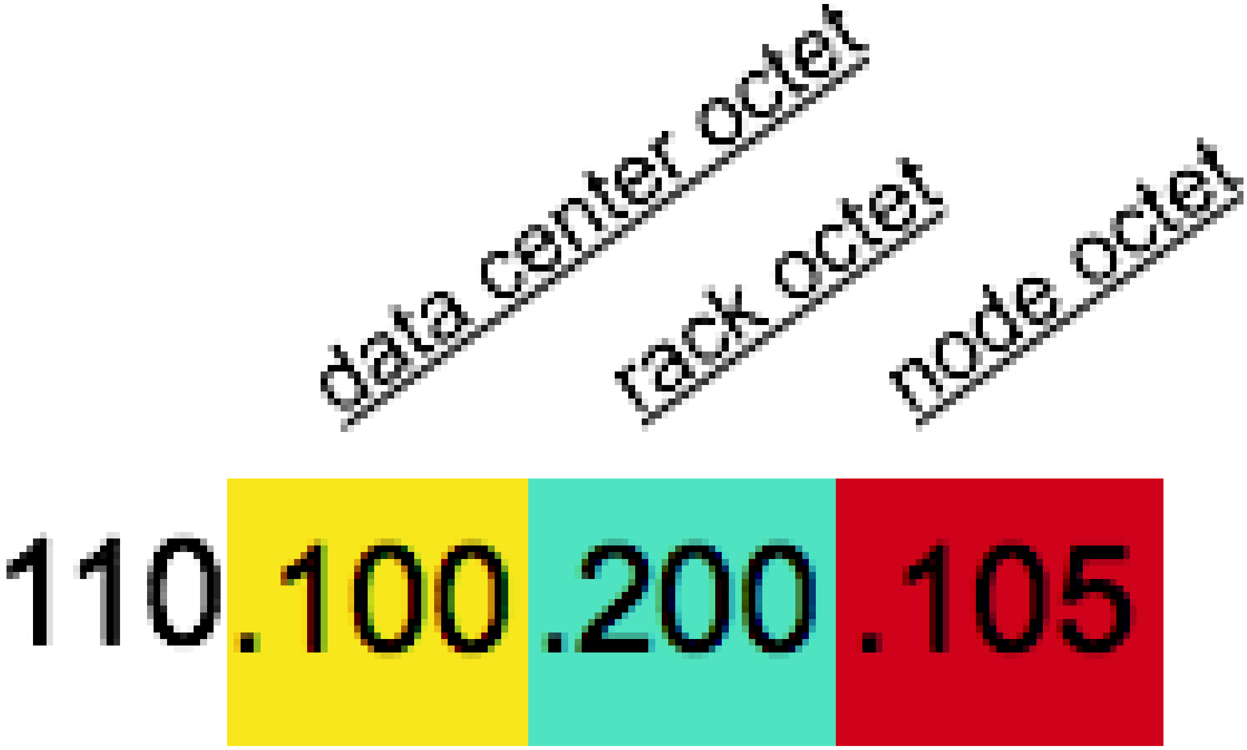
SimpleSnitch

The SimpleSnitch (default) is used only for single-datacenter deployments. It does not recognize datacenter or rack information and can be used only for single-datacenter deployments or single-zone in public clouds. It treats strategy order as proximity, which can improve cache locality when disabling read repair.

Using a SimpleSnitch, you define the keyspace to use SimpleStrategy and specify a replication factor.

RackInferringSnitch

The RackInferringSnitch determines the proximity of nodes by rack and datacenter, which are assumed to correspond to the 3rd and 2nd octet of the node's IP address, respectively. This snitch is best used as an example for writing a custom snitch class (unless this happens to match your deployment conventions).



PropertyFileSnitch

This snitch determines proximity as determined by rack and datacenter. It uses the network details located in the cassandra-topology.properties file. When using this snitch, you can define your datacenter names to be whatever you want. Make sure that the datacenter names correlate to the name of your datacenters in the keyspace definition. Every node in the cluster should be described in the `cassandra-topology.properties` file, and this file should be exactly the same on every node in the cluster.

Procedure

If you had non-uniform IPs and two physical datacenters with two racks in each, and a third logical datacenter for replicating analytics data, the `cassandra-topology.properties` file might look like this:

Note: datacenter and rack names are case-sensitive.

```
# datacenter One

175.56.12.105=DC1:RAC1
175.50.13.200=DC1:RAC1
175.54.35.197=DC1:RAC1

120.53.24.101=DC1:RAC2
120.55.16.200=DC1:RAC2
120.57.102.103=DC1:RAC2

# datacenter Two

110.56.12.120=DC2:RAC1
110.50.13.201=DC2:RAC1
```

```
110.54.35.184=DC2:RAC1

50.33.23.120=DC2:RAC2
50.45.14.220=DC2:RAC2
50.17.10.203=DC2:RAC2

# Analytics Replication Group

172.106.12.120=DC3:RAC1
172.106.12.121=DC3:RAC1
172.106.12.122=DC3:RAC1

# default for unknown nodes
default =DC3:RAC1
```

GossipingPropertyFileSnitch

This snitch is recommended for production. It uses rack and datacenter information for the local node defined in the cassandra-rackdc.properties file and propagates this information to other nodes via gossip.

The configuration for the GossipingPropertyFileSnitch is contained in the `cassandra-rackdc.properties` file.

To configure a node to use GossipingPropertyFileSnitch, edit the `cassandra-rackdc.properties` file as follows:

- Define the datacenter and Rack that include this node. The default settings:

  ```
  dc=DC1
  rack=RAC1
  ```

  Note: datacenter and rack names are case-sensitive.
- To save bandwidth, add the `prefer_local=true` option. This option tells Cassandra to use the local IP address when communication is not across different datacenters.

Migrating from the PropertyFileSnitch to the GossipingPropertyFileSnitch

To allow migration from the PropertyFileSnitch, the GossipingPropertyFileSnitch uses the `cassandra-topology.properties` file when present. Delete the file after the migration is complete. For more information about migration, see Switching snitches on page 147.

Note: The `GossipingPropertyFileSnitch` always loads `cassandra-topology.properties` when that file is present. Remove the file from each node on any new cluster or any cluster migrated from the `PropertyFileSnitch`.

Ec2Snitch

Use the Ec2Snitch for simple cluster deployments on Amazon EC2 where all nodes in the cluster are within a single region.

In EC2 deployments , the region name is treated as the datacenter name and availability zones are treated as racks within a datacenter. For example, if a node is in the us-east-1 region, us-east is the datacenter name and 1 is the rack location. (Racks are important for distributing replicas, but not for datacenter naming.) Because private IPs are used, this snitch does not work across multiple regions.

If you are using only a single datacenter, you do not need to specify any properties.

If you need multiple datacenters, set the dc_suffix options in the cassandra-rackdc.properties file. Any other lines are ignored.

For example, for each node within the us-east region, specify the datacenter in its `cassandra-rackdc.properties` file:

Note: datacenter names are case-sensitive.

- **node0**

  `dc_suffix=_1_cassandra`
- **node1**

  `dc_suffix=_1_cassandra`
- **node2**

  `dc_suffix=_1_cassandra`
- **node3**

  `dc_suffix=_1_cassandra`
- **node4**

  `dc_suffix=_1_analytics`
- **node5**

  `dc_suffix=_1_search`

This results in three datacenters for the region:

```
us-east_1_cassandra
us-east_1_analytics
us-east_1_search
```

Note: The datacenter naming convention in this example is based on the workload. You can use other conventions, such as DC1, DC2 or 100, 200.

Keyspace strategy options

When defining your keyspace strategy options, use the EC2 region name, such as ``us-east``, as your datacenter name.

Ec2MultiRegionSnitch

Use the Ec2MultiRegionSnitch for deployments on Amazon EC2 where the cluster spans multiple regions.

You must configure settings in both the `cassandra.yaml` file and the property file (`cassandra-rackdc.properties`) used by the Ec2MultiRegionSnitch.

Configuring cassandra.yaml for cross-region communication

The Ec2MultiRegionSnitch uses public IP designated in the broadcast_address to allow cross-region connectivity. Configure each node as follows:

1. In the cassandra.yaml, set the listen_address to the *private* IP address of the node, and the broadcast_address to the *public* IP address of the node.

   This allows Cassandra nodes in one EC2 region to bind to nodes in another region, thus enabling multiple datacenter support. For intra-region traffic, Cassandra switches to the private IP after establishing a connection.
2. Set the addresses of the seed nodes in the `cassandra.yaml` file to that of the *public* IP. Private IP are not routable between networks. For example:

   ```
   seeds: 50.34.16.33, 60.247.70.52
   ```

To find the public IP address, from each of the seed nodes in EC2:

```
$ curl http://instance-data/latest/meta-data/public-ipv4
```

Note: Do not make all nodes seeds, see Internode communications (gossip) on page 13.

3. Be sure that the storage_port or ssl_storage_port is open on the public IP firewall.

Configuring the snitch for cross-region communication

In EC2 deployments, the region name is treated as the datacenter name and availability zones are treated as racks within a datacenter. For example, if a node is in the us-east-1 region, us-east is the datacenter name and 1 is the rack location. (Racks are important for distributing replicas, but not for datacenter naming.)

For each node, specify its datacenter in the cassandra-rackdc.properties. The dc_suffix option defines the datacenters used by the snitch. Any other lines are ignored.

In the example below, there are two cassandra datacenters and each datacenter is named for its workload. The datacenter naming convention in this example is based on the workload. You can use other conventions, such as DC1, DC2 or 100, 200. (datacenter names are case-sensitive.)

Region: us-east	Region: us-west

Node and datacenter:
• **node0**
`dc_suffix=_1_cassandra`
• **node1**
`dc_suffix=_1_cassandra`
• **node2**
`dc_suffix=_2_cassandra`
• **node3**
`dc_suffix=_2_cassandra`
• **node4**
`dc_suffix=_1_analytics`
• **node5**
`dc_suffix=_1_search`
This results in four us-east datacenters:
`us-east_1_cassandra`
`us-east_2_cassandra`
`us-east_1_analytics`
`us-east_1_search`
	Node and datacenter:
	• **node0**
	`dc_suffix=_1_cassandra`
	• **node1**
	`dc_suffix=_1_cassandra`
	• **node2**
	`dc_suffix=_2_cassandra`
	• **node3**
	`dc_suffix=_2_cassandra`
	• **node4**
	`dc_suffix=_1_analytics`
	• **node5**
	`dc_suffix=_1_search`
	This results in four us-west datacenters:
	`us-west_1_cassandra`
	`us-west_2_cassandra`
	`us-west_1_analytics`
	`us-west_1_search`

Keyspace strategy options

When defining your keyspace strategy options, use the EC2 region name, such as ``us-east``, as your datacenter name.

Related information

Install locations on page 71

GoogleCloudSnitch

Use the GoogleCloudSnitch for Cassandra deployments on Google Cloud Platform across one or more regions. The region is treated as a datacenter and the availability zones are treated as racks within the datacenter. All communication occurs over private IP addresses within the same logical network.

The region name is treated as the datacenter name and zones are treated as racks within a datacenter. For example, if a node is in the us-central1-a region, us-central1 is the datacenter name and a is the rack location. (Racks are important for distributing replicas, but not for datacenter naming.) This snitch can work across multiple regions without additional configuration.

If you are using only a single datacenter, you do not need to specify any properties.

If you need multiple datacenters, set the dc_suffix options in the cassandra-rackdc.properties file. Any other lines are ignored.

For example, for each node within the us-central1 region, specify the datacenter in its `cassandra-rackdc.properties` file:

Note: datacenter names are case-sensitive.

- **node0**

  ```
  dc_suffix=_a_cassandra
  ```
- **node1**

  ```
  dc_suffix=_a_cassandra
  ```
- **node2**

  ```
  dc_suffix=_a_cassandra
  ```
- **node3**

  ```
  dc_suffix=_a_cassandra
  ```
- **node4**

  ```
  dc_suffix=_a_analytics
  ```
- **node5**

  ```
  dc_suffix=_a_search
  ```

Note: datacenter and rack names are case-sensitive.

CloudstackSnitch

Use the CloudstackSnitch for Apache Cloudstack environments. Because zone naming is free-form in Apache Cloudstack, this snitch uses the widely-used <country> <location> <az> notation.

Database internals

Storage engine

Cassandra uses a storage structure similar to a Log-Structured Merge Tree, unlike a typical relational database that uses a B-Tree. Cassandra avoids reading before writing. Read-before-write, especially in a large distributed system, can result in large latencies in read performance and other problems. For example, two clients read at the same time; one overwrites the row to make update A, and the other overwrites the row to make update B, removing update A. This race condition will result in ambiguous query results - which update is correct?

To avoid using read-before-write for most writes in Cassandra, the storage engine groups inserts and updates in memory, and at intervals, sequentially writes the data to disk in append mode. Once written to disk, the data is immutable and is never overwritten. Reading data involves combining this immutable sequentially-written data to discover the correct query results. You can use Lightweight transactions (LWT) to check the state of the data before writing. However, this feature is recommended only for limited use.

A log-structured engine that avoids overwrites and uses sequential I/O to update data is essential for writing to solid-state disks (SSD) and hard disks (HDD). On HDD, writing randomly involves a higher number of seek operations than sequential writing. The seek penalty incurred can be substantial. Because Cassandra sequentially writes immutable files, thereby avoiding write amplification and disk failure, the database accommodates inexpensive, consumer SSDs extremely well. For many other databases, write amplification is a problem on SSDs.

How Cassandra reads and writes data

To manage and access data in Cassandra, it is important to understand how Cassandra stores data. The hinted handoff feature plus Cassandra conformance and non-conformance to the ACID (atomic, consistent, isolated, durable) database properties are key concepts to understand reads and writes. In Cassandra, consistency refers to how up-to-date and synchronized a row of data is on all of its replicas.

Client utilities and application programming interfaces (APIs) for developing applications for data storage and retrieval are available.

How is data written?

Cassandra processes data at several stages on the write path, starting with the immediate logging of a write and ending in with a write of data to disk:

- Logging data in the commit log
- Writing data to the memtable
- Flushing data from the memtable
- Storing data on disk in SSTables

Logging writes and memtable storage

When a write occurs, Cassandra stores the data in a memory structure called memtable, and to provide configurable durability, it also appends writes to the commit log on disk. The commit log receives every write made to a Cassandra node, and these durable writes survive permanently even if power fails on a node. The memtable is a write-back cache of data partitions that Cassandra looks up by key. The memtable stores writes in sorted order until reaching a configurable limit, and then is flushed.

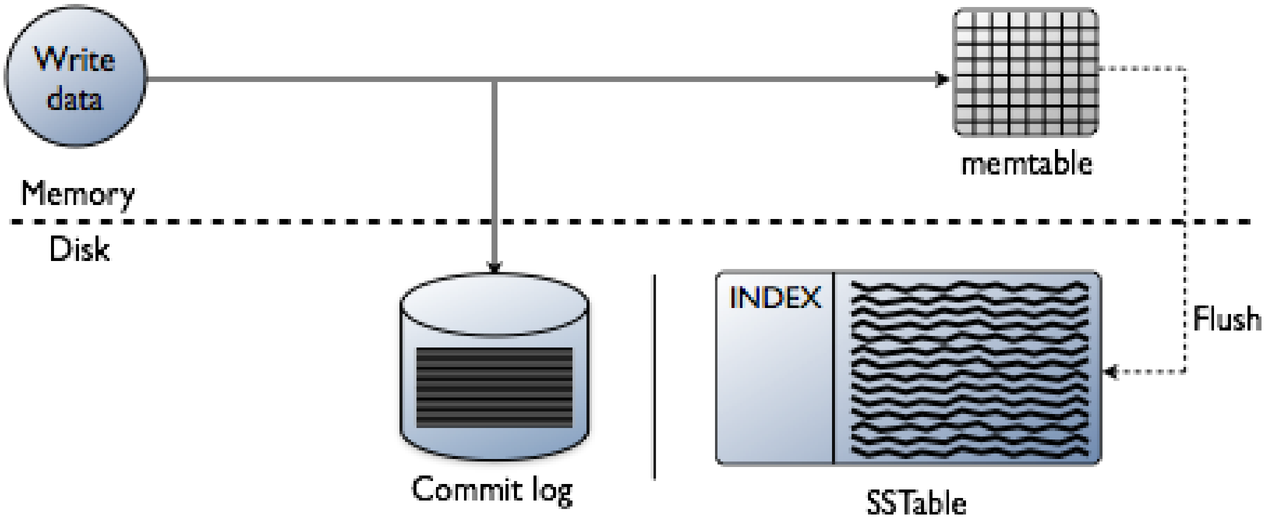
Flushing data from the memtable

To flush the data, Cassandra writes the data to disk, in the memtable-sorted order.. A partition index is also created on the disk that maps the tokens to a location on disk. When the memtable content exceeds the configurable threshold or the commitlog space exceeds the commitlog_total_space_in_mb, the memtable is put in a queue that is flushed to disk. The queue can be configured with the memtable_heap_space_in_mb or memtable_offheap_space_in_mb setting in the cassandra.yaml file. If the data to be flushed exceeds the memtable_cleanup_threshold, Cassandra blocks writes until the next flush succeeds. You can manually flush a table using nodetool flushor nodetool drain (flushes memtables without listening for connections to other nodes). To reduce the commit log replay time, the recommended best practice is to flush the memtable before you restart the nodes. If a node stops working, replaying the commit log restores to the memtable the writes that were there before it stopped.

Data in the commit log is purged after its corresponding data in the memtable is flushed to an SSTable on disk.

Storing data on disk in SSTables

Memtables and SSTables are maintained per table. The commit log is shared among tables. SSTables are immutable, not written to again after the memtable is flushed. Consequently, a partition is typically stored across multiple SSTable files. A number of other SSTable structures exist to assist read operations:



For each SSTable, Cassandra creates these structures:

Data (Data.db)

The SSTable data

Primary Index (Index.db)

Index of the row keys with pointers to their positions in the data file

Bloom filter (Filter.db)

A structure stored in memory that checks if row data exists in the memtable before accessing SSTables on disk

Compression Information (CompressionInfo.db)

A file holding information about uncompressed data length, chunk offsets and other compression information

Statistics (Statistics.db)

Statistical metadata about the content of the SSTable

Digest (Digest.crc32, Digest.adler32, Digest.sha1)

A file holding adler32 checksum of the data file

CRC (CRC.db)

A file holding the CRC32 for chunks in an a uncompressed file.

SSTable Index Summary (SUMMARY.db)

A sample of the partition index stored in memory

SSTable Table of Contents (TOC.txt)

A file that stores the list of all components for the SSTable TOC

Secondary Index (SI_.*.db)

Built-in secondary index. Multiple SIs may exist per SSTable

The SSTables are files stored on disk. The naming convention for SSTable files has changed with Cassandra 2.2 and later to shorten the file path. The data files are stored in a data directory that varies with installation. For each keyspace, a directory within the data directory stores each table. For example, `/data/data/ks1/cf1-5be396077b811e3a3ab9dc4b9ac088d/la-1-big-Data.db` represents a data file. ks1 represents the keyspace name to distinguish the keyspace for streaming or bulk loading data. A hexadecimal string, 5be396077b811e3a3ab9dc4b9ac088d in this example, is appended to table names to represent unique table IDs.

Cassandra creates a subdirectory for each table, which allows you to symlink a table to a chosen physical drive or data volume. This provides the capability to move very active tables to faster media, such as SSDs for better performance, and also divides tables across all attached storage devices for better I/O balance at the storage layer.

How is data maintained?

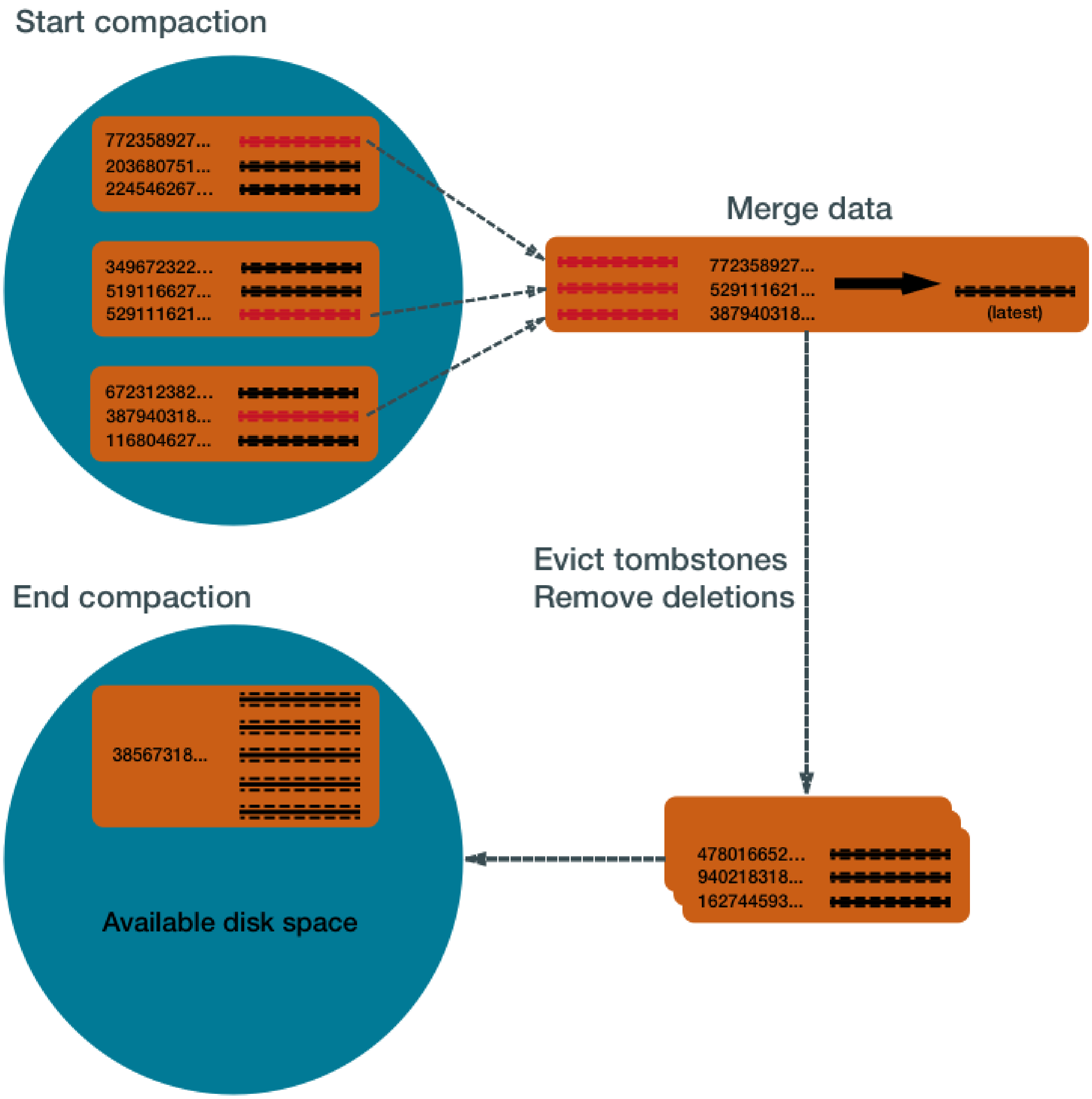
The Cassandra write process stores data in files called SSTables. SSTables are immutable. Instead of overwriting existing rows with inserts or updates, Cassandra writes new timestamped versions of the inserted or updated data in new SSTables. Cassandra does not perform deletes by removing the deleted data: instead, Cassandra marks it with tombstones.

Over time, Cassandra may write many versions of a row in different SSTables. Each version may have a unique set of columns stored, and a different timestamp. This distribution of data could require Cassandra to access more and more SSTables to retrieve a complete row.

To keep the database healthy, Cassandra periodically merges SSTables and discards old data. This process is called compaction.

Compaction

Compaction works on a collection of SSTables. From these SSTables, compaction collects all versions of each unique row and assembles one complete row, using the most up-to-date version (by timestamp) of each of the row's columns. This merge process is performant. Because rows are sorted by partition key within each SSTable, the merge process does not use random I/O. These new versions of these rows are written to a new SSTable. The old versions, along with any rows that are ready for deletion, are left in the old SSTables. These old SSTable files are deleted as soon as pending reads finish using them.



Compaction causes a temporary spike in disk space usage and disk I/O while old and new SSTables co-exist. As it completes, compaction frees up disk space occupied by old SSTables. It improves read performance by incrementally replacing old SSTables with compacted SSTables. Cassandra can read data directly from the new SSTable even before it finishes writing, instead of waiting for the entire compaction process to finish.

As Cassandra processes writes and reads, it replaces the old SSTables with new SSTables in the page cache. The process of caching the new SSTable, while directing reads away from the old one, is incremental — it does not cause a the dramatic cache miss. Cassandra provides predictable high performance even under heavy load.

Compaction strategies

Cassandra supports different compaction strategies, which control how which SSTables are chosen for compaction, and how the compacted rows are sorted into new SSTables. Each strategy has its own strengths. The sections that follow explain each of Cassandra's compaction strategies.

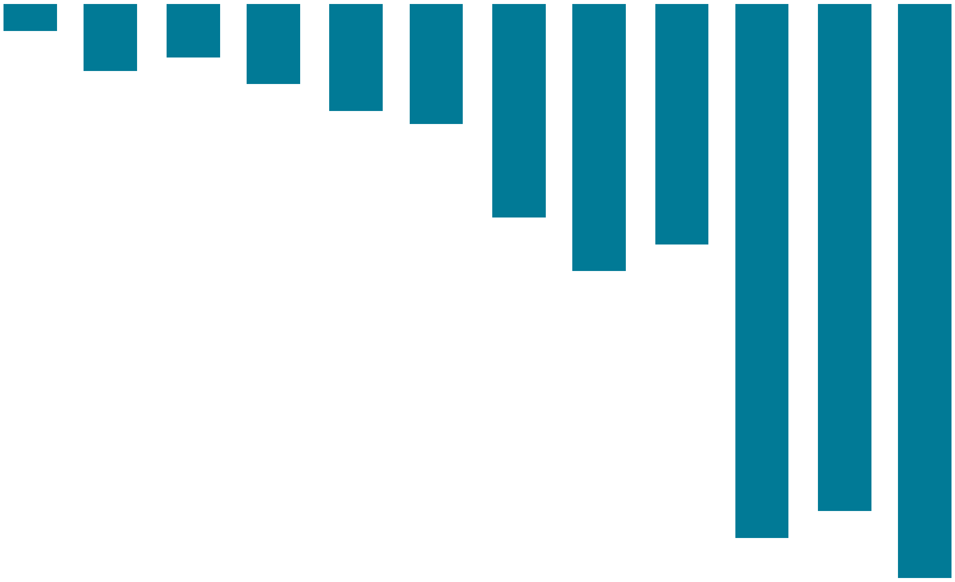
Although each of the following sections starts with a generalized recommendation, many factors complicate the choice of a compaction strategy. See Which compaction strategy is best?.

SizeTieredCompactionStrategy (STCS)

Recommended for write-intensive workloads.

The SizeTieredCompactionStrategy (STCS) initiates compaction when Cassandra has accumulated a set number (default: 4) of similar-sized SSTables. STCS merges these SSTables into one larger SSTable. As these larger SSTables accumulate, STCS merges these into even larger SSTables. At any given time, several SSTables of varying sizes are present.

Figure: Size tiered compaction after many inserts



While STCS works well to compact a write-intensive workload, it makes reads slower because the merge-by-size process does not group data by rows. This makes it more likely that versions of a particular row may be spread over many SSTables. Also, STCS does not evict deleted data predictably becauseits trigger for compaction is SSTable size, and SSTables might not grow quickly enough to merge and evict old data. As the largest SSTables grow in size, the amount of disk space needed for both the new and old SSTables simultaneously during STCS compaction can outstrip a typical amount of disk space on a node.

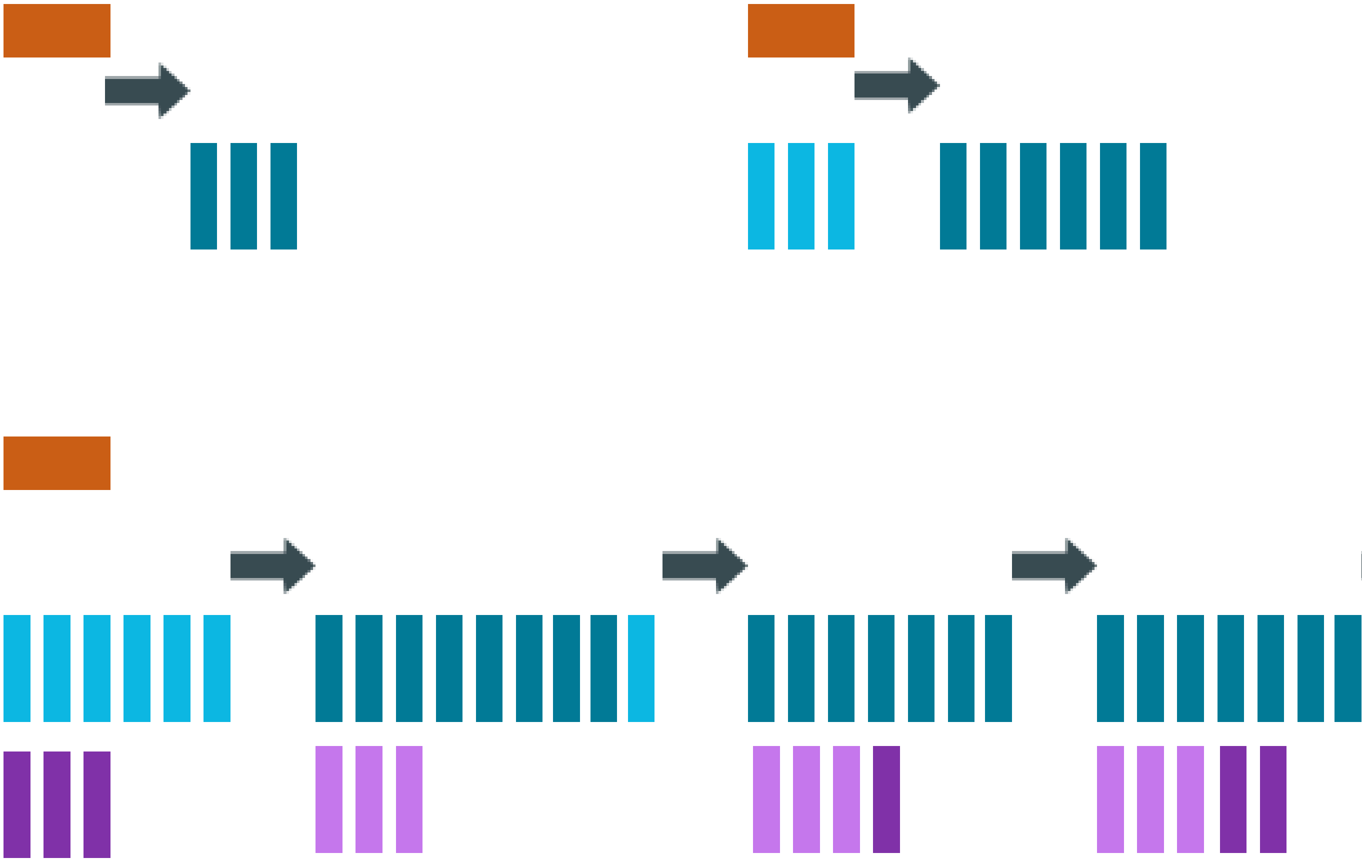
- **Pros:** Compacts write-intensive workload very well.
- **Cons:** Can hold onto stale data too long. Amount of memory needed increases over time.

LeveledCompactionStrategy (LCS)

Recommended for read-intensive workloads.

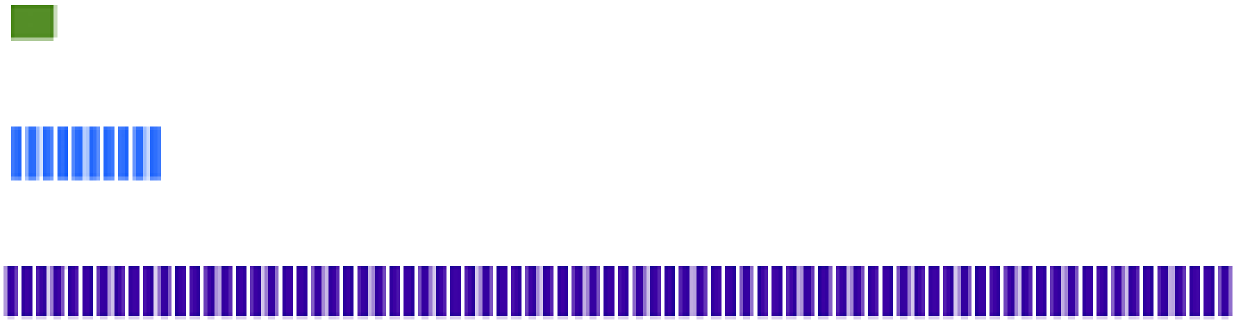
The LeveledCompactionStrategy (LCS) alleviates some of the read operation issues with STCS. This strategy works with a series of levels. First, data in mmtables is flushed to SSTables in the first level (L0). LCS compaction merges these first SSTables with larger SSTables in level L1.

Figure: Leveled compaction — adding SSTables



The SSTables in levels greater than L1 are merged into SSTables with a size greater than or equal to `sstable_size_in_mb` (default: 160 MB). If a L1 SSTable stores data of a partition that is larger than L2, LCS moves the SSTable past L2 to the next level up.

Figure: Leveled compaction after many inserts



In each of the levels above L0, LCS creates SSTables that are about the same size. Each level is 10X the size of the last level, so level L1 has 10X as many SSTables as L0, and level L2 has 100X as many. If the result of the compaction is more than 10 SSTables in level L1, the excess SSTables are moved to level L2.

The LCS compaction process guarantees that the SSTables within each level starting with L1 have non-overlapping data. For many reads, this process enables Cassandra to retrieve all the required data from only one or two SSTables. In fact, 90% of all reads can be satisfied from one SSTable. Since LCS does not compact L0 tables, however, resource-intensive reads involving many L0 SStables may still occur.

At levels beyond L0, LCS requires less disk space for compacting — generally, 10X the fixed size of the SSTable. Obsolete data is evicted more often, so deleted data uses smaller portions of the SSTables on disk. However, LCS compaction operations take place more often and place more I/O burden on the node. For write-intensive workloads, the payoff of using this strategy is generally not worth the performance loss to I/O operations. In many cases, tests of LCS-configured tables reveal I/O saturation on writes and compactions.

Note: Cassandra bypasses compaction operations when bootstrapping a new node using LCS into a cluster. The original data is moved directly to the correct level because there is no existing data, so no partition overlap per level is present. For more information, see Apache Cassandra 2.2 - Bootstrapping Performance Improvements for Leveled Compaction.

- **Pros:** Disk requirements are easier to predict. Read operation latency is more predictable. Stale data is evicted more frequently.
- **Cons:** Much higher I/O utilization impacting operation latency

DateTieredCompactionStrategy (DTCS)

Recommended for time series and expiring TTL workloads.

The DateTieredCompactionStrategy (DTCS) is similar to STCS. But instead of compacting based on SSTable size, DTCS compacts based on SSTable age. (Each column in an SSTable is marked with the timestamp at write time. As the *age* of an SSTable, DTCS uses the oldest timestamp of any column the SSTable contains.)

Configuring the DTCS time window ensures that new and old data are not mixed in merged SSTables. In fact, using Time-To-Live (TTL) timestamps, DTCS can eject whole SSTables containing expired data. This strategy often generates similar-sized SSTables if time series data is ingested at a steady rate.

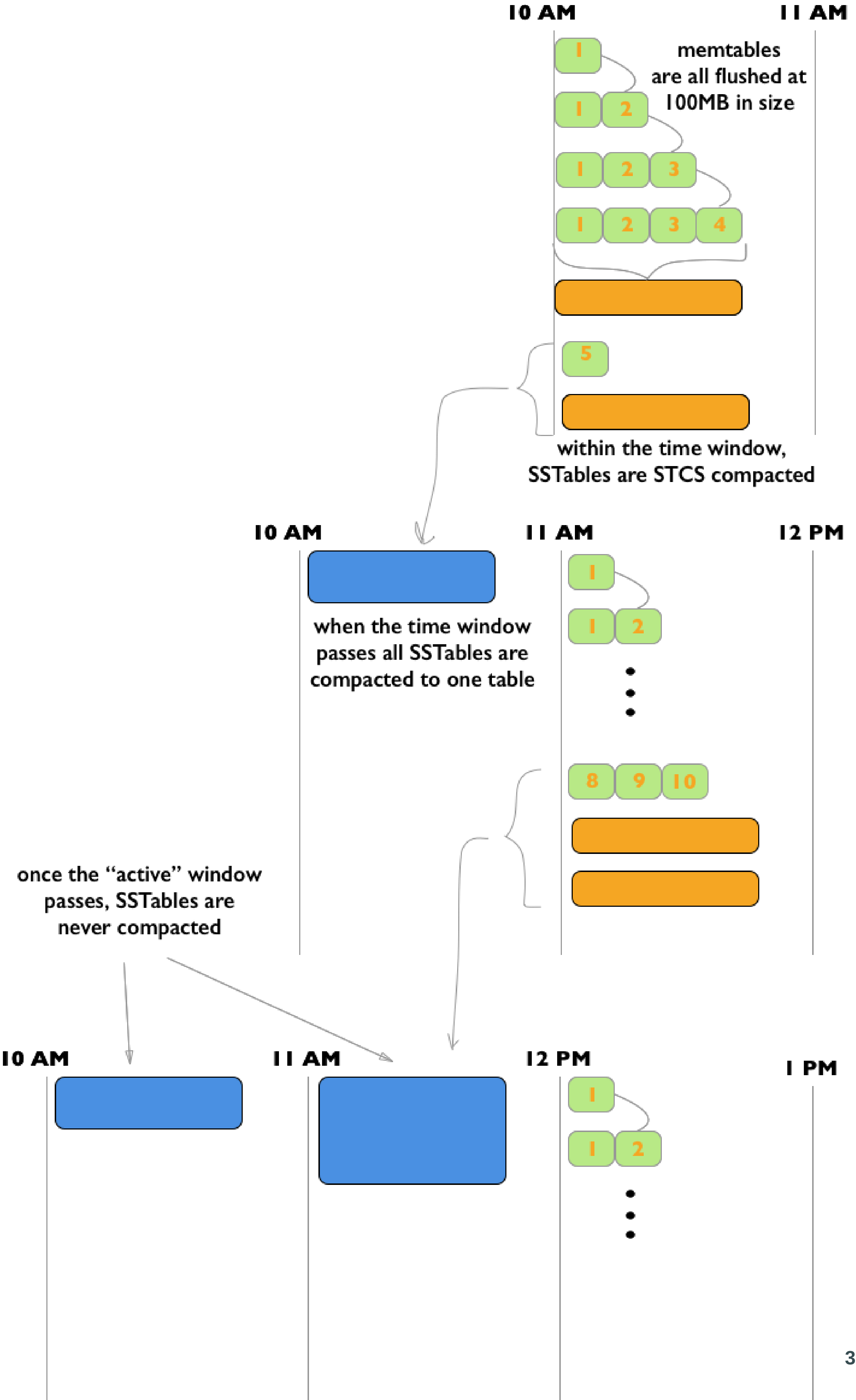
DTCS compacts SSTables into larger tables, as in STC, when the system accumulates a configurable number of SSTables within a configurable time interval. However, DTCS skips compacting SSTables that reach a configurable age. This logic reduces the number of times data is rewritten. Queries that ask for data in a particular last time interval, such as an hour, can be executed very efficiently on DTCS-compacted SSTables (particularly if the requested time interval is coordinated with the configured interval for compaction).

One usecase that can cause difficulty with this strategy is out-of-order writing. For example, an operation that writes a timestamped record with a past timestamp. Read repairs can inject out-of-order timestamps, so be sure to turn off read repairs when using DTCS.

- **Pros**: Specifically designed for time series data, stored in tables that use the default TTL. DTCS is a better choice when fine-tuning is required to meet space-related SLAs.
- **Cons**: Insertion of records out of time order (by repairs or hint replaying) can increase latency or cause errors. In some cases, it may be necessary to turn off read repair and carefully test and control the use of TIMESTAMP options in BATCH, DELETE, INSERT and UPDATE CQL commands.

TimeWindowCompactionStrategy (TWCS)

The TimeWindowCompactionStrategy (TWCS) is similar to DTCS. TWCS groups SSTables using a series of time windows or buckets. During its operation, TWCS applies SizeTieredCompactionStrategy to uncompacted SSTables in the most recent time window. TWCS compacts SSTables that fall into the next time window into a single SSTable. SSTables that fall into older time windows are not subject to further compaction. At the next compaction, TWCS handles the newest SSTables, and older SSTables, in the same way.


10 AM
11 AM
1
memtables are all flushed at 100MB in size
1
2
1
2
3
1
2
3
4
5
within the time window, SSTables are STCS compacted
10 AM
11 AM
12 PM
1
when the time window passes all SSTables are compacted to one table
1
2
8
9
10
once the "active" window passes, SSTables are never compacted
10 AM
11 AM
12 PM
1 PM
1
1
2

As the figure shows, from 10 to 11AM, the memtables are flushed from memory into 100MB SSTables. These SSTables are compacted into larger SSTables using STCS. At 11, all these SSTables are compacted into a single SSTable, and never compacted again by TWCS. At 12, the new SSTables created between 11 and 12 are compacted using STCS, and at the end of the time window the TWCS compaction repeats. Notice that each TWCS time window contains varying amounts of data.

Note: For an animated explanation, see the Datastax Academy Time Window Compaction Strategy video.

The TWCS configuration has two main property settings:

- compaction_window_unit: time unit used to define the bucket size (milliseconds, seconds, hours, and so on)
- compaction_window_size: how many units per bucket (1,2,3, and so on)

The configuration for the above example: `compaction_window_unit = ‘minutes’,compaction_window_size = 60`

Pros: Used for time series data, stored in tables that use the default TTL for all data. Simpler configuration than that of DTCS.

Cons: Less fine-tuned configuration is possible than with DTCS.

Which compaction strategy is best?

To implement the best compaction strategy:

1. Review your application's requirements.
2. Configure the table to use the most appropriate strategy.
3. Test the compaction strategies against your data.

The following questions are based on the experiences of Cassandra developers and users with the strategies described above.

Does your table process time series data?

If so, your best choices are DTCS or TWCS. For details, read the descriptions on this page.

If your table is not focused on time series data, the choice becomes more complicated The following questions introduce other considerations that may guide your choice.

Does your table handle more reads than writes, or more writes than reads?

LCS is a good choice if your table processes twice as many reads as writes or more – especially randomized reads. If the proportion of reads to writes is closer, the performance hit exacted by LCS may not be worth the benefit. Be aware that LCS can be quickly overwhelmed by a high volume of writes.

Does the data in your table change often?

One advantage of LCS is that it keeps related data in a small set of SSTables. If your data is *immutable* or not subject to frequent upserts, STCS accomplishes the same type of grouping without the LCS performance hit.

Do you require predictable levels of read and write activity?

LCS keeps the SSTables within predictable sizes and numbers. For example, if your table's read/write ratio is small, and it is expected to conform to a Service Level Agreements (SLAs) for reads, it may be worth taking the write performance penalty of LCS in order to keep read rates and latency at predictable levels. And you may be able to overcome this write penalty through *horizontal scaling* — adding more nodes.

Will your table be populated by a batch process?

On both batch reads and batch writes, STCS performs better than LCS. The batch process causes little or no fragmentation, so the benefits of LCS are not realized. And batch processes can overwhelm LCS-configured tables.

Does your system have limited disk space?

LCS handles disk space more efficiently than STCS: it requires about 10% *headroom* in addition to the space occupied by the data is handles. STCS and DTCS generally require more — in some cases. 50% more than the data space.

Is your system reaching its limits for I/O?

LCS is significantly more I/O intensive than DTCS or STCS. Switching to LCS may introduce extra I/O load that offsets the advantages.

Testing compaction strategies

Suggestions for determining which compaction strategy is best for your system:

- Create a three-node cluster using one of the compaction strategies, stress test the cluster using cassandra-stress, and measure the results.
- Set up a node on your existing cluster and use Cassandra's write survey mode to sample live data. See What’s new in Cassandra 1.1: live traffic sampling.

Configuring and running compaction

Set the compaction strategy for a table in the parameters for the `CREATE TABLE` or `ALTER TABLE` command. For details, see Table properties.

You can start compaction manually using the nodetool compact command.

More information about compaction

The following blog posts provide more information from developers that have tested compaction strategies:

- When to Use Leveled Compaction
- Leveled compaction in Apache Cassandra
- DateTieredCompactionStrategy: Notes from the Field
- Date-Tiered Compaction in Cassandra
- DateTieredCompactionStrategy: Compaction for Time Series Data.
- What delays a tombstone purge when using LCS in Cassandra

How is data updated?

Cassandra treats each new row as an upsert: if the new row has the same primary key as that of an existing row, Cassandra processes it as an update to the existing row.

During a write, Cassandra adds each new row to the database without checking on whether a duplicate record exists. This policy makes it possible that many versions of the same row may exist in the database. For more details about writes, see How is data written?

Periodically, the rows stored in memory are streamed to disk into structures called SSTables. At certain intervals, Cassandra compacts smaller SSTables into larger SSTables. If Cassandra encounters two or more versions of the same row during this process, Cassandra only writes the most recent version to the new SSTable. After compaction, Cassandra drops the original SSTables, deleting the outdated rows.

Most Cassandra installations store replicas of each row on two or more nodes. Each node performs compaction independently. This means that even though out-of-date versions of a row have been dropped from one node, they may still exist on another node.

This is why Cassandra performs another round of comparisons during a read process. When a client requests data with a particular primary key, Cassandra retrieves many versions of the row from one or more replicas. The version with the most recent timestamp is the only one returned to the client ("last-write-wins").

Note: Some database operations may only write partial updates of a row, so some versions of a row may include some columns, but not all. During a compaction or write, Cassandra assembles a complete version of each row from the partial updates, using the most recent version of each column.

How is data deleted?

Cassandra's processes for deleting data are designed to improve performance, and to work with Cassandra's built-in properties for data distribution and fault-tolerance.

Cassandra treats a delete as an insert or upsert. The data being added to the partition in the DELETE command is a deletion marker called a tombstone. The tombstones go through Cassandra's write path, and are written to SSTables on one or more nodes. The key difference feature of a tombstone: it has a built-in expiration date/time. At the end of its expiration period (for details see below) the tombstone is deleted as part of Cassandra's normal compaction process.

You can also mark a Cassandra record (row or column) with a time-to-live value. After this amount of time has ended, Cassandra marks the record with a tombstone, and handles it like other tombstoned records.

Deletion in a distributed system

In a multi-node cluster, Cassandra can store replicas of the same data on two or more nodes. This helps prevent data loss, but it complicates the delete process. If a node receives a delete for data it stores locally, the node tombstones the specified record and tries to pass the tombstone to other nodes containing replicas of that record. But if one replica node is unresponsive at that time, it does not receive the tombstone immediately, so it still contains the pre-delete version of the record. If the tombstoned record has already been deleted from the rest of the cluster befor that node recovers, Cassandra treats the record on the recovered node as new data, and propagates it to the rest of the cluster. This kind of deleted but persistent record is called a zombie.

To prevent the reappearance of zombies, Cassandra gives each tombstone a *grace period*. The purpose of the grace period is to give unresponsive nodes time to recover and process tombstones normally. If a client writes a new update to the tombstoned record during the grace period, Cassandra overwrites the tombstone. If a client sends a read for that record during the grace period, Cassandra disregards the tombstone and retrieves the record from other replicas if possible.

When an unresponsive node recovers, Cassandra uses hinted handoff to replay the database mutations the node missed while it was down. Cassandra does not replay a mutation for a tombstoned record during its grace period. But if the node does not recover until after the grace period ends, Cassandra may miss the deletion.

After the tombstone's grace period ends, Cassandra deletes the tombstone during compaction.

The grace period for a tombstone is set by the property gc_grace_seconds. Its default value is 864000 seconds (ten days). Each table can have its own value for this property.

More about Cassandra deletes

Details:

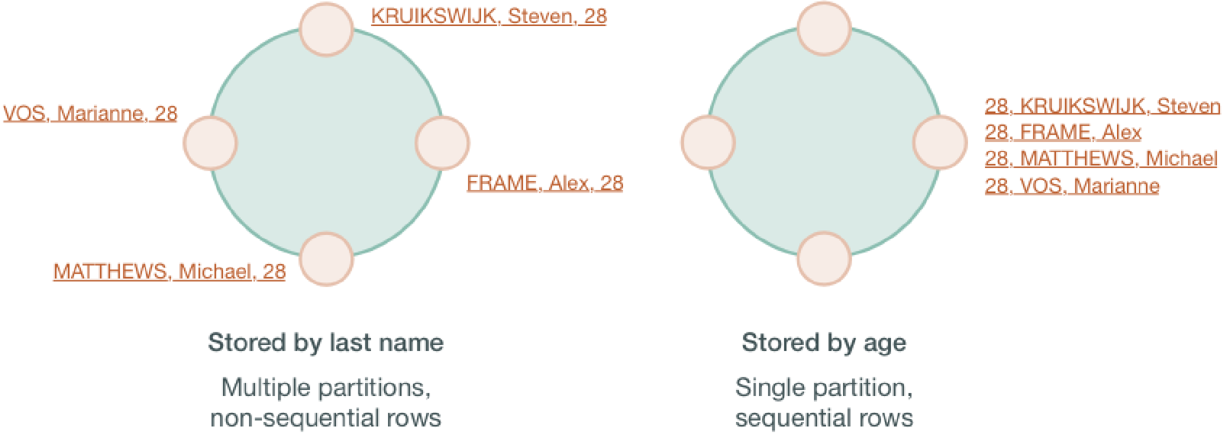
- The expiration date/time for a tombstone is the date/time of its creation plus the value of the table property *gc_grace_seconds*.
- Cassandra also supports Batch data insertion and updates. This procedure also introduces the danger of replaying a record insertion after that record has been removed from the rest of the cluster. Cassandra does not replay a batched mutation for a tombstoned record that is still within its grace period.
- On a single-node cluster, you can set *gc_grace_seconds* to `0` (zero).
- To completely prevent the reappearance of zombie records, run nodetool repair on a node after it recovers, and on each table every *gc_grace_seconds*.

- If all records in a table are given a TTL at creation, and all are allowed to expire and not deleted manually, it is not necessary to run nodetool repair for that table on a regular basis.
- If you use the SizeTieredCompactionStrategy or DateTieredCompactionStrategy, you can delete tombstones immediately by manually starting the compaction process.

  CAUTION: If you force compaction, Cassandra may create one very large SSTable from all the data. Cassandra will not trigger another compaction for a long time. The data in the SSTable created during the forced compaction can grow very stale during this long period of non-compaction.
- Cassandra allows you to set a *default_time_to_live* property for an entire table. Columns and rows marked with regular TTLs are processed as described above; but when a record exceeds the table-level TTL, Cassandra deletes it immediately, without tombstoning or compaction.
- Cassandra supports immediate deletion through the DROP KEYSPACE and DROP TABLE statements.

How are indexes stored and updated?

Secondary indexes are used to filter a table for data stored in non-primary key columns. For example, a table storing cyclist names and ages using the last name of the cyclist as the primary key might have a secondary index on the age to allow queries by age. Querying to match a non-primary key column is an anti-pattern, as querying should always result in a continuous slice of data retrieved from the table.



If the table rows are stored based on last names, the table may be spread across several partitions stored on different nodes. Queries based on a particular range of last names, such as all cyclists with the last name `Matthews` will retrieve sequential rows from the table, but a query based on the age, such as all cyclists who are `28`, will require all nodes to be queried for a value. Non-primary keys play no role in ordering the data in storage, thus querying for a particular value of a non-primary key column results in scanning all partitions. Scanning all partitions generally results in a prohibitive read latency, and is not allowed.

Secondary indexes can be built for a column in a table. These indexes are stored locally on each node in a hidden table and built in a background process. If a secondary index is used in a query that is not restricted to a particular partition key, the query will have prohibitive read latency because all nodes will be queried. A query with these parameters is only allowed if the query option `ALLOW FILTERING` is used. This option is not appropriate for production environments. If a query includes both a partition key condition and a secondary index column condition, the query will be successful because the query can be directed to a single node partition.

This technique, however, does not guarantee trouble-free indexing, so know when and when not to use an index. In the example shown above, an index on the age could be used, but a better solution is to create a materialized view or additional table that is ordered by age.

As with relational databases, keeping indexes up to date uses processing time and resources, so unnecessary indexes should be avoided. When a column is updated, the index is updated as well. If the old column value still exists in the memtable, which typically occurs when updating a small set of rows repeatedly, Cassandra removes the corresponding obsolete index entry; otherwise, the old entry remains to be purged by compaction. If a read sees a stale index entry before compaction purges it, the reader thread invalidates it.

How is data read?

To satisfy a read, Cassandra must combine results from the active memtable and potentially multiple SSTables.

Cassandra processes data at several stages on the read path to discover where the data is stored, starting with the data in the memtable and finishing with SSTables:

- Check the memtable
- Check row cache, if enabled
- Checks Bloom filter
- Checks partition key cache, if enabled
- Goes directly to the compression offset map if a partition key is found in the partition key cache, or checks the partition summary if not

  If the partition summary is checked, then the partition index is accessed
- Locates the data on disk using the compression offset map
- Fetches the data from the SSTable on disk

Figure: Read request flow

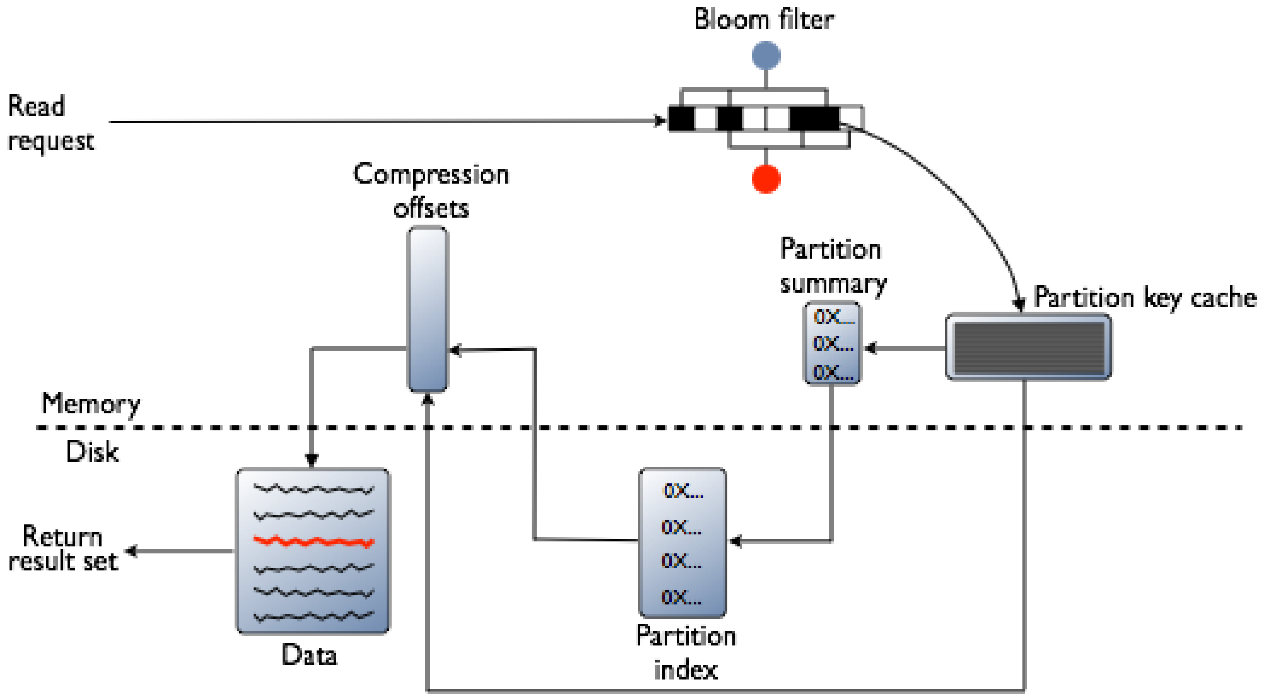
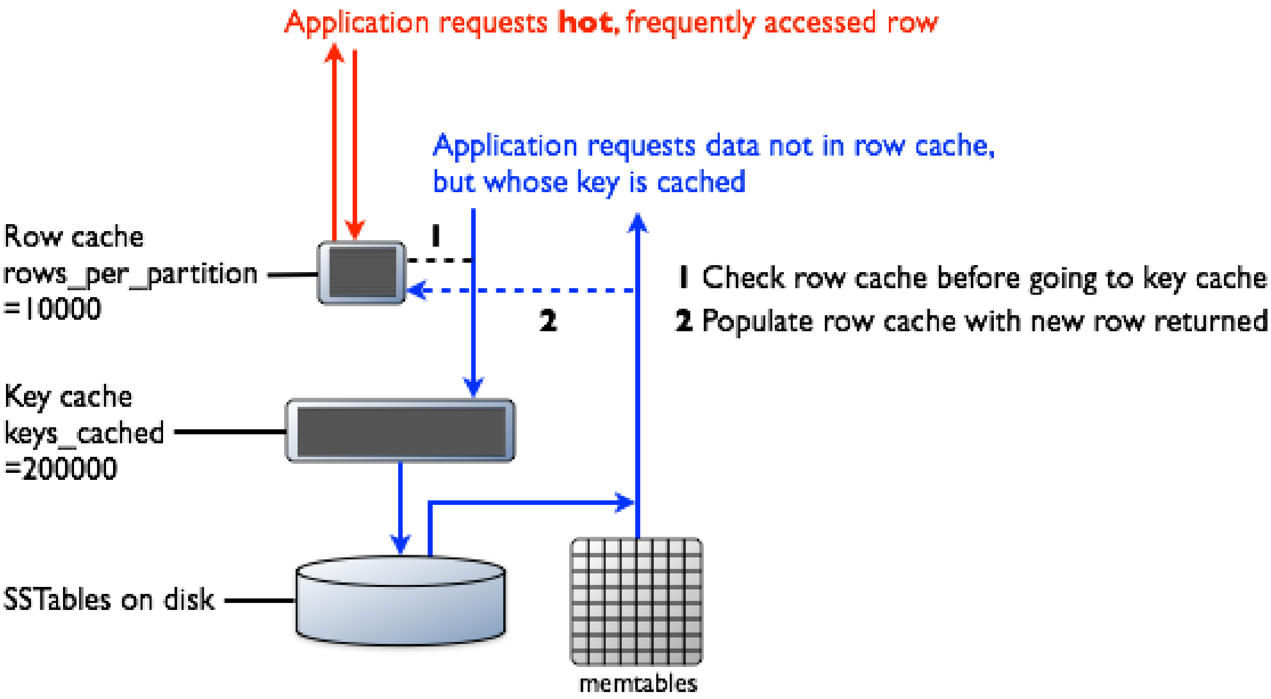


Figure: Row cache and Key cache request flow


Application requests hot, frequently accessed row
Application requests data not in row cache, but whose key is cached
Row cache
rows_per_partition
=10000
1
2
1 Check row cache before going to key cache
2 Populate row cache with new row returned
Key cache
keys_cached
=200000
SSTables on disk
memtables

Memtable

If the memtable has the desired partition data, then the data is read and then merged with the data from the SSTables. The SSTable data is accessed as shown in the following steps.

Row Cache

Typical of any database, reads are fastest when the most in-demand data fits into memory. The operating system page cache is best at improving performance, although the row cache can provide some improvement for very read-intensive operations, where read operations are 95% of the load. Row cache is contra-indicated for write-intensive operations. The row cache, if enabled, stores a subset of the partition data stored on disk in the SSTables in memory. In Cassandra 2.2 and later, it is stored in fully off-heap memory using a new implementation that relieves garbage collection pressure in the JVM. The subset stored in the row cache use a configurable amount of memory for a specified period of time. The row cache uses LRU (least-recently-used) eviction to reclaim memory when the cache has filled up.

The row cache size is configurable, as is the number of rows to store. Configuring the number of rows to be stored is a useful feature, making a "Last 10 Items" query very fast to read. If row cache is enabled, desired partition data is read from the row cache, potentially saving two seeks to disk for the data. The rows stored in row cache are frequently accessed rows that are merged and saved to the row cache from the SSTables as they are accessed. After storage, the data is available to later queries. The row cache is not write-through. If a write comes in for the row, the cache for that row is invalidated and is not cached again until the row is read. Similarly, if a partition is updated, the entire partition is evicted from the cache. When the desired partition data is not found in the row cache, then the Bloom filter is checked.

Bloom Filter

First, Cassandra checks the Bloom filter to discover which SSTables are likely to have the request partition data. The Bloom filter is stored in off-heap memory. Each SSTable has a Bloom filter associated with it. A Bloom filter can establish that a SSTable does not contain certain partition data. A Bloom filter can also find the likelihood that partition data is stored in a SSTable. It speeds up the process of partition key lookup by narrowing the pool of keys. However, because the Bloom filter is a probabilistic function, it can result in false positives. Not all SSTables identified by the Bloom filter will have data. If the Bloom filter does not rule out an SSTable, Cassandra checks the partition key cache

The Bloom filter grows to approximately 1-2 GB per billion partitions. In the extreme case, you can have one partition per row, so you can easily have billions of these entries on a single machine. The Bloom filter is tunable if you want to trade memory for performance.

Partition Key Cache

The partition key cache, if enabled, stores a cache of the partition index in off-heap memory. The key cache uses a small, configurable amount of memory, and each "hit" saves one seek during the read operation. If a partition key is found in the key cache can go directly to the compression offset map to find the compressed block on disk that has the data. The partition key cache functions better once warmed, and can greatly improve over the performance of cold-start reads, where the key cache doesn't yet have or has purged the keys stored in the key cache. It is possible to limit the number of partition keys saved in the key cache, if memory is very limited on a node. If a partition key is not found in the key cache, then the partition summary is searched.

The partition key cache size is configurable, as are the number of partition keys to store in the key cache.

Partition Summary

The partition summary is an off-heap in-memory structure that stores a sampling of the partition index. A partition index contains all partition keys, whereas a partition summary samples every X keys, and maps the location of every Xth key's location in the index file. For example, if the partition summary is set to sample every 20 keys, it will store the location of the first key as the beginning of the SSTable file, the 20th key and its location in the file, and so on. While not as exact as knowing the location of the partition key, the partition summary can shorten the scan to find the partition data location. After finding the range of possible partition key values, the partition index is searched.

By configuring the sample frequency, you can trade memory for performance, as the more granularity the partition summary has, the more memory it will use. The sample frequency is changed using the index interval property in the table definition. A fixed amount of memory is configurable using the index_summary_capacity_in_mb property, and defaults to 5% of the heap size.

Partition Index

The partition index resides on disk and stores an index of all partition keys mapped to their offset. If the partition summary has been checked for a range of partition keys, now the search passes to the partition index to seek the location of the desired partition key. A single seek and sequential read of the columns over the passed-in range is performed. Using the information found, the partition index now goes to the compression offset map to find the compressed block on disk that has the data. If the partition index must be searched, two seeks to disk will be required to find the desired data.

Compression offset map

The compression offset map stores pointers to the exact location on disk that the desired partition data will be found. It is stored in off-heap memory and is accessed by either the partition key cache or the partition index. The desired compressed partition data is fetched from the correct SSTable(s) once the compression offset map identifies the disk location. The query receives the result set.

Note: Within a partition, all rows are not equally expensive to query. The very beginning of the partition (the first rows, clustered by your key definition) is slightly less expensive to query because there is no need to consult the partition-level index.

The compression offset map grows to 1-3 GB per terabyte compressed. The more you compress data, the greater number of compressed blocks you have and the larger the compression offset table. Compression is enabled by default even though going through the compression offset map consumes CPU resources. Having compression enabled makes the page cache more effective, and typically, almost always pays off.

How do write patterns affect reads?

It is important to consider how the write operations will affect the read operations in the cluster. The type of compaction strategy Cassandra performs on your data is configurable and can significantly affect read performance. Using the SizeTieredCompactionStrategy or DateTieredCompactionStrategy tends to cause data fragmentation when rows are frequently updated. The LeveledCompactionStrategy (LCS) was designed to prevent fragmentation under this condition.

Data consistency

How are consistent read and write operations handled?

Consistency refers to how up-to-date and synchronized all replicas of a row of Cassandra data are at any given moment. Ongoing repair operations in Cassandra ensure that all replicas of a row will eventually be consistent. Repairs work to decrease the variability in replica data, but constant data traffic through a widely distributed system can lead to inconsistency (stale data) at any given time. Cassandra is a AP system according to the CAP theorem, providing high availability and partition tolerance. Cassandra does have flexibility in its configuration, though, and can perform more like a CP (consistent and partition tolerant) system according to the CAP theorem, depending on the application requirements. Two consistency features are tunable consistency and linearizable consistency.

Tunable consistency

To ensure that Cassandra can provide the proper levels of consistency for its reads and writes, Cassandra extends the concept of eventual consistency by offering tunable consistency. You can tune Cassandra's consistency level per-operation, or set it globally for a cluster or datacenter. You can vary the consistency for individual read or write operations so that the data returned is more or less consistent, as required by the client application. This allows you to make Cassandra act more like a CP (consistent and partition tolerant) or AP (highly available and partition tolerant) system according to the CAP theorem, depending on the application requirements.

Note: It is not possible to "tune" Cassandra into a completely CA system. See You Can't Sacrifice Partition Tolerance for a more detailed discussion.

There is a tradeoff between operation latency and consistency: higher consistency incurs higher latency, lower consistency permits lower latency. You can control latency by tuning consistency.

The consistency level determines the number of replicas that need to acknowledge the read or write operation success to the client application. For read operations, the read consistency level specifies how many replicas must respond to a read request before returning data to the client application. If a read operation reveals inconsistency among replicas, Cassandra initiates a read repair to update the inconsistent data.

For write operations, the write consistency level specified how many replicas must respond to a write request before the write is considered successful. Even at low consistency levels, Cassandra writes to all replicas of the partition key, including replicas in other datacenters. The write consistency level just specifies when the coordinator can report to the client application that the write operation is considered completed. Write operations will use hinted handoffs to ensure the writes are completed when replicas are down or otherwise not responsive to the write request.

Typically, a client specifies a consistency level that is less than the replication factor specified by the keyspace. Another common practice is to write at a consistency level of QUORUM and read at a consistency level of QUORUM. The choices made depend on the client application's needs, and Cassandra provides maximum flexibility for application design.

Linearizable consistency

In ACID terms, linearizable consistency (or serial consistency) is a serial (immediate) isolation level for lightweight transactions. Cassandra does not use employ traditional mechanisms like locking or transactional dependencies when concurrently updating multiple rows or tables.

However, some operations must be performed in sequence and not interrupted by other operations. For example, duplications or overwrites in user account creation can have serious consequences. Situations like race conditions (two clients updating the same record) can introduce inconsistency across replicas. Writing with high consistency does nothing to reduce this. You can apply linearizable consistency to a unique identifier, like the userID or email address, although is not required for all aspects of the user's account. Serial operations for these elements can be implemented in Cassandra with the Paxos consensus protocol, which uses a quorum-based algorithm. Lightweight transactions can be implemented without the need for a master database or two-phase commit process.

Lightweight transaction write operations use the serial consistency level for Paxos consensus and the regular consistency level for the write to the table. For more information, see Lightweight Transactions.

Calculating consistency

Reliability of read and write operations depends on the consistency used to verify the operation. Strong consistency can be guaranteed when the following condition is true:

```
R + W > N
```

where

- R is the consistency level of read operations
- W is the consistency level of write operations
- N is the number of replicas

If the replication factor is 3, then the consistency level of the reads and writes combined must be at least 4. For example, read operations using 2 out of 3 replicas to verify the value, and write operations using 2 out of 3 replicas to verify the value will result in strong consistency. If fast write operations are required, but strong consistency is still desired, the write consistency level is lowered to 1, but now read operations have to verify a matched value on all 3 replicas. Writes will be fast, but reads will be slower.

Eventual consistency occurs if the following condition is true:

```
R + W =< N
```

where

- R is the consistency level of read operations
- W is the consistency level of write operations
- N is the number of replicas

If the replication factor is 3, then the consistency level of the reads and writes combined are 3 or less. For example, read operations using QUORUM (2 out of 3 replicas) to verify the value, and write operations using ONE (1 out of 3 replicas) to do fast writes will result in eventual consistency. All replicas will receive the data, but read operations are more vulnerable to selecting data before all replicas write the data.

Additional consistency examples:

- You do a write at ONE, the replica crashes one second later. The other messages are not delivered. The data is lost.
- You do a write at ONE, and the operation times out. Future reads can return the old or the new value. You will not know the data is incorrect.
- You do a write at ONE, and one of the other replicas is down. The node comes back online. The application will get old data from that node until the node gets the correct data or a read repair occurs.

- You do a write at QUORUM, and then a read at QUORUM. One of the replicas dies. You will always get the correct data.

How are Cassandra transactions different from RDBMS transactions?

Cassandra does not use RDBMS ACID transactions with rollback or locking mechanisms, but instead offers atomic, isolated, and durable transactions with eventual/tunable consistency that lets the user decide how strong or eventual they want each transaction's consistency to be.

As a non-relational database, Cassandra does not support joins or foreign keys, and consequently does not offer consistency in the ACID sense. For example, when moving money from account A to B the total in the accounts does not change. Cassandra supports atomicity and isolation at the row-level, but trades transactional isolation and atomicity for high availability and fast write performance. Cassandra writes are durable.

Atomicity

In Cassandra, a write operation is atomic at the partition level, meaning the insertions or updates of two or more rows in the same partition are treated as one write operation. A delete operation is also atomic at the partition level.

For example, if using a write consistency level of QUORUM with a replication factor of 3, Cassandra will replicate the write to all nodes in the cluster and wait for acknowledgement from two nodes. If the write fails on one of the nodes but succeeds on the other, Cassandra reports a failure to replicate the write on that node. However, the replicated write that succeeds on the other node is not automatically rolled back.

Cassandra uses client-side timestamps to determine the most recent update to a column. The latest timestamp always wins when requesting data, so if multiple client sessions update the same columns in a row concurrently, the most recent update is the one seen by readers.

Isolation

Cassandra write and delete operations are performed with full row-level isolation. This means that a write to a row within a single partition on a single node is only visible to the client performing the operation – the operation is restricted to this scope until it is complete. All updates in a batch operation belonging to a given partition key have the same restriction. However, a Batch operation is not isolated if it includes changes to more than one partition.

Durability

Writes in Cassandra are durable. All writes to a replica node are recorded both in memory and in a commit log on disk before they are acknowledged as a success. If a crash or server failure occurs before the memtables are flushed to disk, the commit log is replayed on restart to recover any lost writes. In addition to the local durability (data immediately written to disk), the replication of data on other nodes strengthens durability.

You can manage the local durability to suit your needs for consistency using the commitlog_sync option in the cassandra.yaml file. Set the option to either periodic or batch.

How do I accomplish lightweight transactions with linearizable consistency?

Distributed databases present a unique challenge when data must be strictly read and written in sequential order. In transactions for creating user accounts or transferring money, race conditions between two potential writes must be regulated to ensure that one write precedes the other. In Cassandra, the Paxos consensus protocol is used to implement lightweight transactions that can handle concurrent operations.

The Paxos protocol is implemented in Cassandra with linearizable consistency, that is sequential consistency with real-time constraints. Linearizable consistency ensures transaction isolation at a level similar to the serializable level offered by RDBMSs. This type of transaction is known as compare and set (CAS); replica data is compared and any data found to be out of date is set to the most consistent value. In Cassandra, the process combines the Paxos protocol with normal read and write operations to accomplish the compare and set operation.

The Paxos protocol is implemented as a series of phases:

1. Prepare/Promise
2. Read/Results
3. Propose/Accept
4. Commit/Acknowledge

These phases are actions that take place between a proposer and acceptors. Any node can be a proposer, and multiple proposers can be operating at the same time. For simplicity, this description will use only one proposer. A proposer prepares by sending a message to a quorum of acceptors that includes a proposal number. Each acceptor promises to accept the proposal if the proposal number is the highest they have received. Once the proposer receives a quorum of acceptors who promise, the value for the proposal is read from each acceptor and sent back to the proposer. The proposer figures out which value to use and proposes the value to a quorum of the acceptors along with the proposal number. Each acceptor accepts the proposal with a certain number if and only if the acceptor is not already promised to a proposal with a high number. The value is committed and acknowledged as a Cassandra write operation if all the conditions are met.

These four phases require four round trips between a node proposing a lightweight transaction and any cluster replicas involved in the transaction. Performance will be affected. Consequently, reserve lightweight transactions for situations where concurrency must be considered.

Lightweight transactions will block other lightweight transactions from occurring, but will not stop normal read and write operations from occurring. Lightweight transactions use a timestamping mechanism different than for normal operations and mixing LWTs and normal operations can result in errors. If lightweight transactions are used to write to a row within a partition, only lightweight transactions for both read and write operations should be used. This caution applies to all operations, whether individual or batched. For example, the following series of operations can fail:

```
DELETE ...
INSERT .... IF NOT EXISTS
SELECT ....
```

The following series of operations will work:

```
DELETE ... IF EXISTS
INSERT .... IF NOT EXISTS
SELECT .....
```

Reads with linearizable consistency

A SERIAL consistency level allows reading the current (and possibly uncommitted) state of data without proposing a new addition or update. If a SERIAL read finds an uncommitted transaction in progress, Cassandra performs a read repair as part of the commit.

How do I discover consistency level performance?

Before changing the consistency level on read and write operations, discover how your CQL commands are performing using the `TRACING` command in CQL. Using `cqlsh`, you can vary the consistency level and trace read and write operations. The tracing output includes latency times for the operations.

The CQL documentation includes a tutorial comparing consistency levels.

How is the consistency level configured?

Consistency levels in Cassandra can be configured to manage availability versus data accuracy. You can configure consistency on a cluster, datacenter, or per individual read or write operation. Consistency among participating nodes can be set globally and also controlled on a per-operation basis. Within `cqlsh`, use `CONSISTENCY`, to set the consistency level for all queries in the current `cqlsh` session. For programming client applications, set the consistency level using an appropriate driver. For example, using the Java driver, call `QueryBuilder.insertInto` with `setConsistencyLevel` to set a per-insert consistency level.

The consistency level defaults to `ONE` for all write and read operations.

Write consistency levels

This table describes the write consistency levels in strongest-to-weakest order.

Table: Write Consistency Levels

Level	Description	Usage
`ALL`	A write must be written to the commit log and memtable on all replica nodes in the cluster for that partition.	Provides the highest consistency and the lowest availability of any other level.
`EACH_QUORUM`	Strong consistency. A write must be written to the commit log and memtable on a quorum of replica nodes in *each*datacenters.	Used in multiple datacenter clusters to strictly maintain consistency at the same level in each datacenter. For example, choose this level if you want a read to fail when a datacenter is down and the `QUORUM` cannot be reached on that datacenter.
`QUORUM`	A write must be written to the commit log and memtable on a quorum of replica nodes across *all* datacenters.	Used in either single or multiple datacenter clusters to maintain strong consistency across the cluster. Use if you can tolerate some level of failure.
`LOCAL_QUORUM`	Strong consistency. A write must be written to the commit log and memtable on a quorum of replica nodes in the same datacenter as the coordinator. Avoids latency of inter-datacenter communication.	Used in multiple datacenter clusters with a rack-aware replica placement strategy, such as NetworkTopologyStrategy, and a properly configured snitch. Use to maintain consistency locally (within the single datacenter). Can be used with SimpleStrategy.
`ONE`	A write must be written to the commit log and memtable of at least one replica node.	Satisfies the needs of most users because consistency requirements are not stringent.
`TWO`	A write must be written to the commit log and memtable of at least two replica nodes.	Similar to `ONE`.
`THREE`	A write must be written to the commit log and memtable of at least three replica nodes.	Similar to `TWO`.
`LOCAL_ONE`	A write must be sent to, and successfully acknowledged by, at least one replica node in the local datacenter.	In a multiple datacenter clusters, a consistency level of `ONE` is often desirable, but cross-DC traffic is not. `LOCAL_ONE` accomplishes this. For security and quality reasons, you can use this consistency level in an offline datacenter to prevent

Level	Description	Usage
		automatic connection to online nodes in other datacenters if an offline node goes down.
ANY	A write must be written to at least one node. If all replica nodes for the given partition key are down, the write can still succeed after a hinted handoff has been written. If all replica nodes are down at write time, an ANY write is not readable until the replica nodes for that partition have recovered.	Provides low latency and a guarantee that a write never fails. Delivers the lowest consistency and highest availability.

Read consistency levels

This table describes read consistency levels in strongest-to-weakest order.

Table: Read Consistency Levels

Level	Description	Usage
ALL	Returns the record after all replicas have responded. The read operation will fail if a replica does not respond.	Provides the highest consistency of all levels and the lowest availability of all levels.
EACH_QUORUM	Not supported for reads.	
QUORUM	Returns the record after a quorum of replicas from all datacenters has responded.	Used in either single or multiple datacenter clusters to maintain strong consistency across the cluster. Ensures strong consistency if you can tolerate some level of failure.
LOCAL_QUORUM	Returns the record after a quorum of replicas in the current datacenter as the coordinator has reported. Avoids latency of inter-datacenter communication.	Used in multiple datacenter clusters with a rack-aware replica placement strategy (NetworkTopologyStrategy) and a properly configured snitch. Fails when using SimpleStrategy.
ONE	Returns a response from the closest replica, as determined by the snitch. By default, a read repair runs in the background to make the other replicas consistent.	Provides the highest availability of all the levels if you can tolerate a comparatively high probability of stale data being read. The replicas contacted for reads may not always have the most recent write.
TWO	Returns the most recent data from two of the closest replicas.	Similar to ONE.
THREE	Returns the most recent data from three of the closest replicas.	Similar to TWO.
LOCAL_ONE	Returns a response from the closest replica in the local datacenter.	Same usage as described in the table about write consistency levels.
SERIAL	Allows reading the current (and possibly uncommitted) state of data without proposing a new addition or update. If a SERIAL read finds an uncommitted transaction in progress, it will commit the	To read the latest value of a column after a user has invoked a lightweight transaction to write to the column, use SERIAL. Cassandra then checks the inflight lightweight transaction for updates and, if found, returns the latest data.

Level	Description	Usage
	transaction as part of the read. Similar to QUORUM.	
LOCAL_SERIAL	Same as SERIAL, but confined to the datacenter. Similar to LOCAL_QUORUM.	Used to achieve linearizable consistency for lightweight transactions.

How QUORUM is calculated

The QUORUM level writes to the number of nodes that make up a quorum. A quorum is calculated, and then rounded down to a whole number, as follows:

```
quorum = (sum_of_replication_factors / 2) + 1
```

The sum of all the replication_factor settings for each datacenter is the sum_of_replication_factors.

```
sum_of_replication_factors = datacenter1_RF + datacenter2_RF + . . . +
 datacentern_RF
```

Examples:

- Using a replication factor of 3, a quorum is 2 nodes. The cluster can tolerate 1 replica down.
- Using a replication factor of 6, a quorum is 4. The cluster can tolerate 2 replicas down.
- In a two datacenter cluster where each datacenter has a replication factor of 3, a quorum is 4 nodes. The cluster can tolerate 2 replica nodes down.
- In a five datacenter cluster where two datacenters have a replication factor of 3 and three datacenters have a replication factor of 2, a quorum is 7 nodes.

The more datacenters, the higher number of replica nodes need to respond for a successful operation.

Similar to QUORUM, the LOCAL_QUORUM level is calculated based on the replication factor of the same datacenter as the coordinator node. That is, even if the cluster has more than one datacenter, the quorum is calculated only with local replica nodes.

In EACH_QUORUM, every datacenter in the cluster must reach a quorum based on that datacenter's replication factor in order for the read or write request to succeed. That is, for every datacenter in the cluster a quorum of replica nodes must respond to the coordinator node in order for the read or write request to succeed.

Configuring client consistency levels

You can use a cqlsh command, CONSISTENCY, to set the consistency level for queries in the current cqlsh session. For programming client applications, set the consistency level using an appropriate driver. For example, call QueryBuilder.insertInto with a setConsistencyLevel argument using the Java driver.

How is the serial consistency level configured?

Serial consistency levels in Cassandra can be configured to manage lightweight transaction isolation. Lightweight transactions have two consistency levels defined. The serial consistency level defines the consistency level of the serial phase, or Paxos phase, of lightweight transactions. The learn phase, which defines what read operations will be guaranteed to complete immediately if lightweight writes are occurring uses a normal consistency level. The serial consistency level is ignored for any query that is not a conditional update.

Serial consistency levels

Table: Serial Consistency Levels

Level	Description	Usage
`SERIAL`	Achieves linearizable consistency for lightweight transactions by preventing unconditional updates.	This consistency level is only for use with lightweight transaction. Equivalent to QUORUM.
`LOCAL_SERIAL`	Same as SERIAL but confined to the datacenter. A conditional write must be written to the commit log and memtable on a quorum of replica nodes in the same datacenter.	Same as SERIAL but used to maintain consistency locally (within the single datacenter). Equivalent to LOCAL_QUORUM.

How are read requests accomplished?

There are three types of read requests that a coordinator can send to a replica:

- A direct read request
- A digest request
- A background read repair request

In a direct read request, the coordinator node contacts one replica node. Then the coordinator sends a digest request to a number of replicas determined by the consistency level specified by the client. The digest request checks the data in the replica node to make sure it is up to date. Then the coordinator sends a digest request to all remaining replicas. If any replica nodes have out of date data, a background read repair request is sent. Read repair requests ensure that the requested row is made consistent on all replicas.

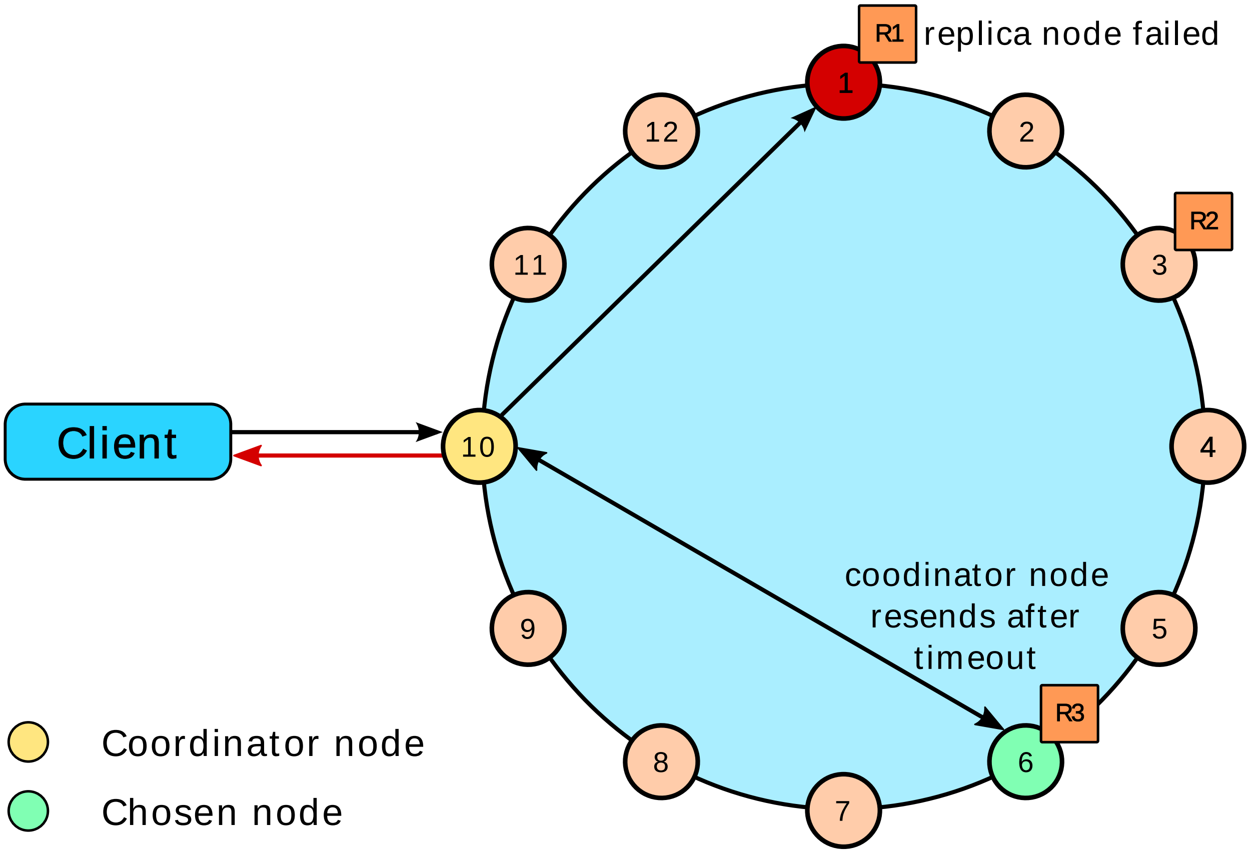
For a digest request the coordinator first contacts the replicas specified by the consistency level. The coordinator sends these requests to the replicas that currently responds the fastest. The contacted nodes respond with a digest of the requested data; if multiple nodes are contacted, the rows from each replica are compared in memory for consistency. If they are not consistent, the replica having the most recent data (based on the timestamp) is used by the coordinator to forward the result back to the client. To ensure that all replicas have the most recent version of the data, read repair is carried out to update out-of-date replicas.

For illustrated examples of read requests, see Examples of read consistency levels on page 49`.

Rapid read protection using speculative_retry

Rapid read protection allows Cassandra to still deliver read requests when the originally selected replica nodes are either down or taking too long to respond. If the table has been configured with the `speculative_retry` property, the coordinator node for the read request will retry the request with another replica node if the original replica node exceeds a configurable timeout value to complete the read request.

Figure: Recovering from replica node failure with rapid read protection



Examples of read consistency levels

The following diagrams show examples of read requests using these consistency levels:

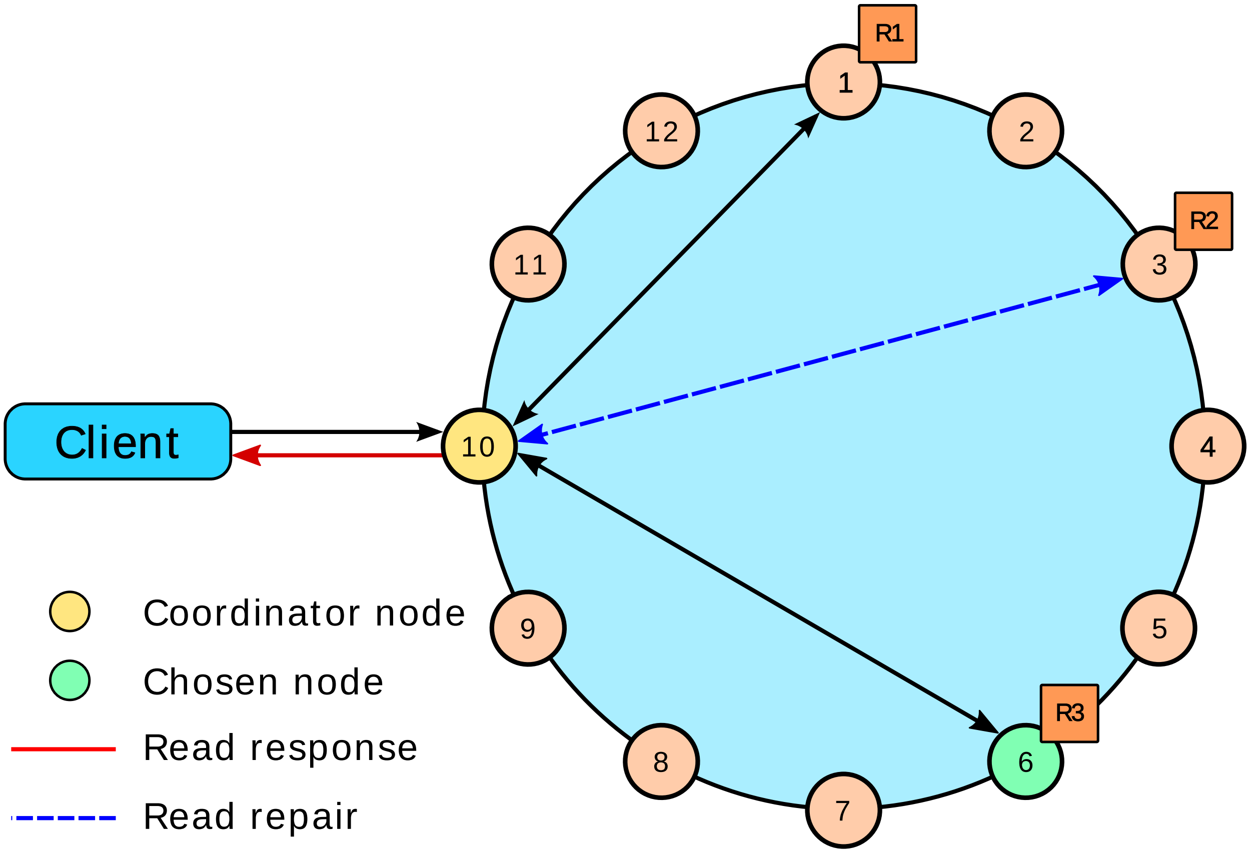
- QUORUM in a single datacenter
- ONE in a single datacenter
- QUORUM in two datacenters
- LOCAL_QUORUM in two datacenters
- ONE in two datacenters
- LOCAL_ONE in two datacenters

Rapid read protection diagram shows how the speculative retry table property affects consistency.

A single datacenter cluster with a consistency level of `QUORUM`

In a single datacenter cluster with a replication factor of 3, and a read consistency level of `QUORUM`, 2 of the 3 replicas for the given row must respond to fulfill the read request. If the contacted replicas have different versions of the row, the replica with the most recent version will return the requested data. In the background, the third replica is checked for consistency with the first two, and if needed, a read repair is initiated for the out-of-date replicas.

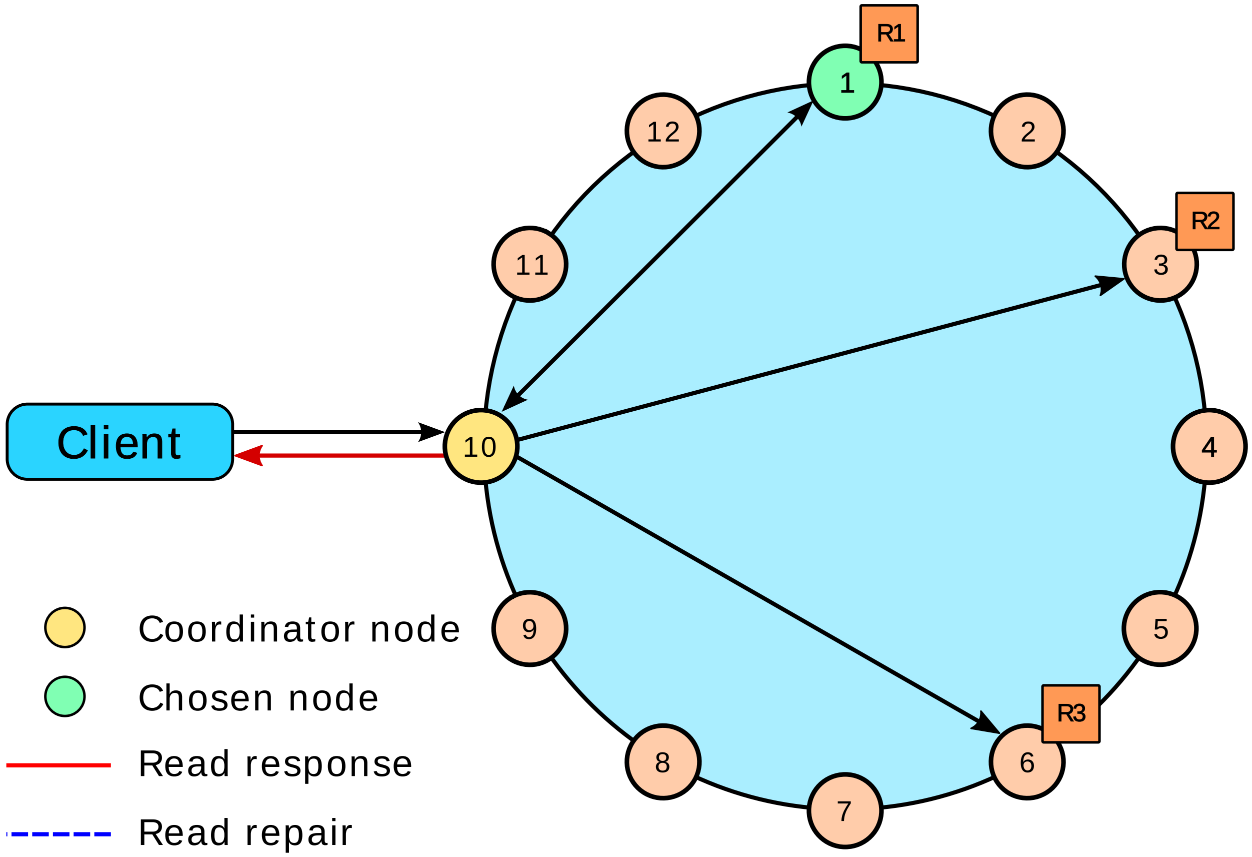
Figure: Single datacenter cluster with 3 replica nodes and consistency set to QUORUM



A single datacenter cluster with a consistency level of ONE

In a single datacenter cluster with a replication factor of 3, and a read consistency level of `ONE`, the closest replica for the given row is contacted to fulfill the read request. In the background a read repair is potentially initiated, based on the `read_repair_chance` setting of the table, for the other replicas.

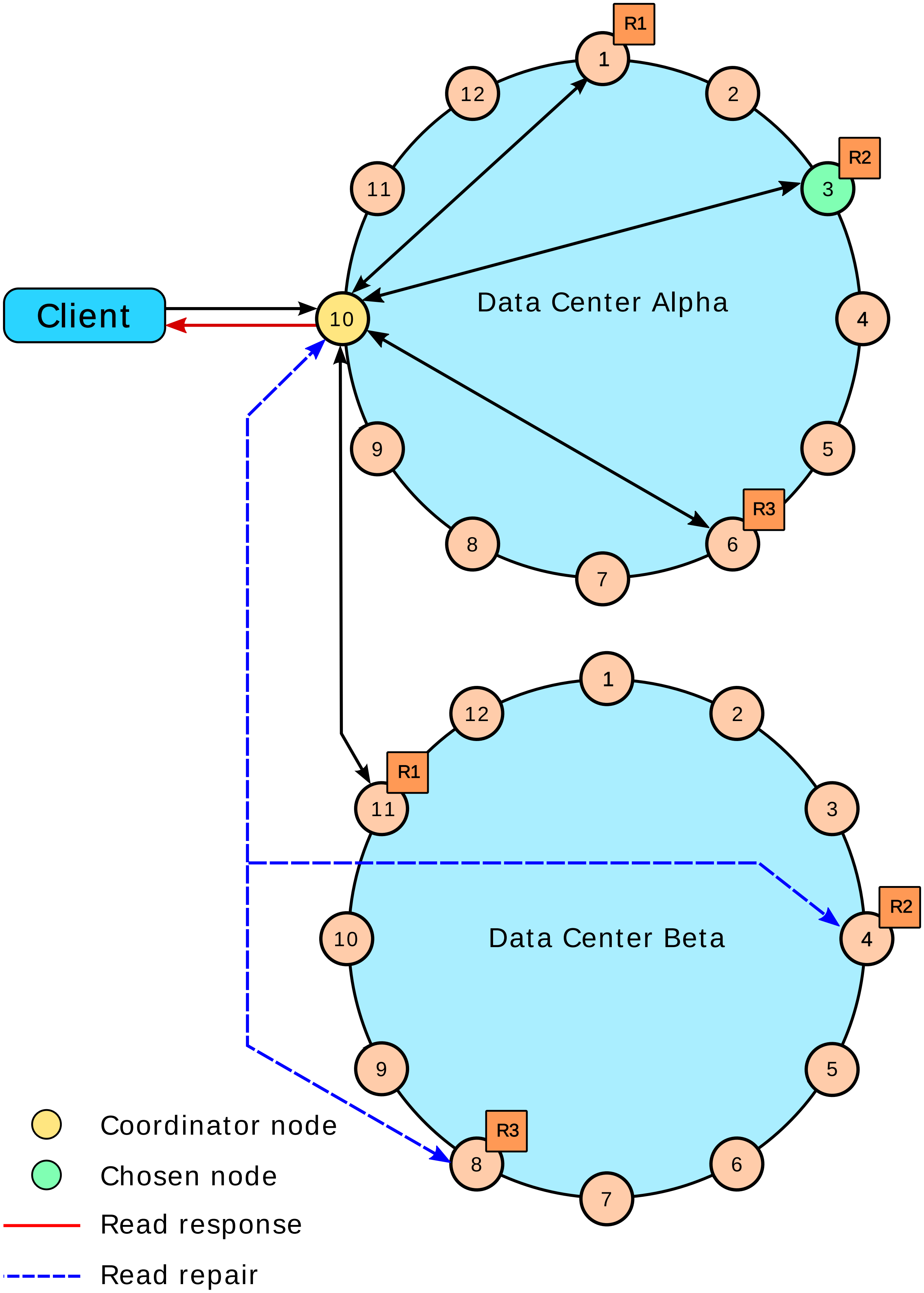
Figure: Single datacenter cluster with 3 replica nodes and consistency set to ONE



A two datacenter cluster with a consistency level of `QUORUM`

In a two datacenter cluster with a replication factor of 3, and a read consistency of `QUORUM`, 4 replicas for the given row must respond to fulfill the read request. The 4 replicas can be from any datacenter. In the background, the remaining replicas are checked for consistency with the first four, and if needed, a read repair is initiated for the out-of-date replicas.

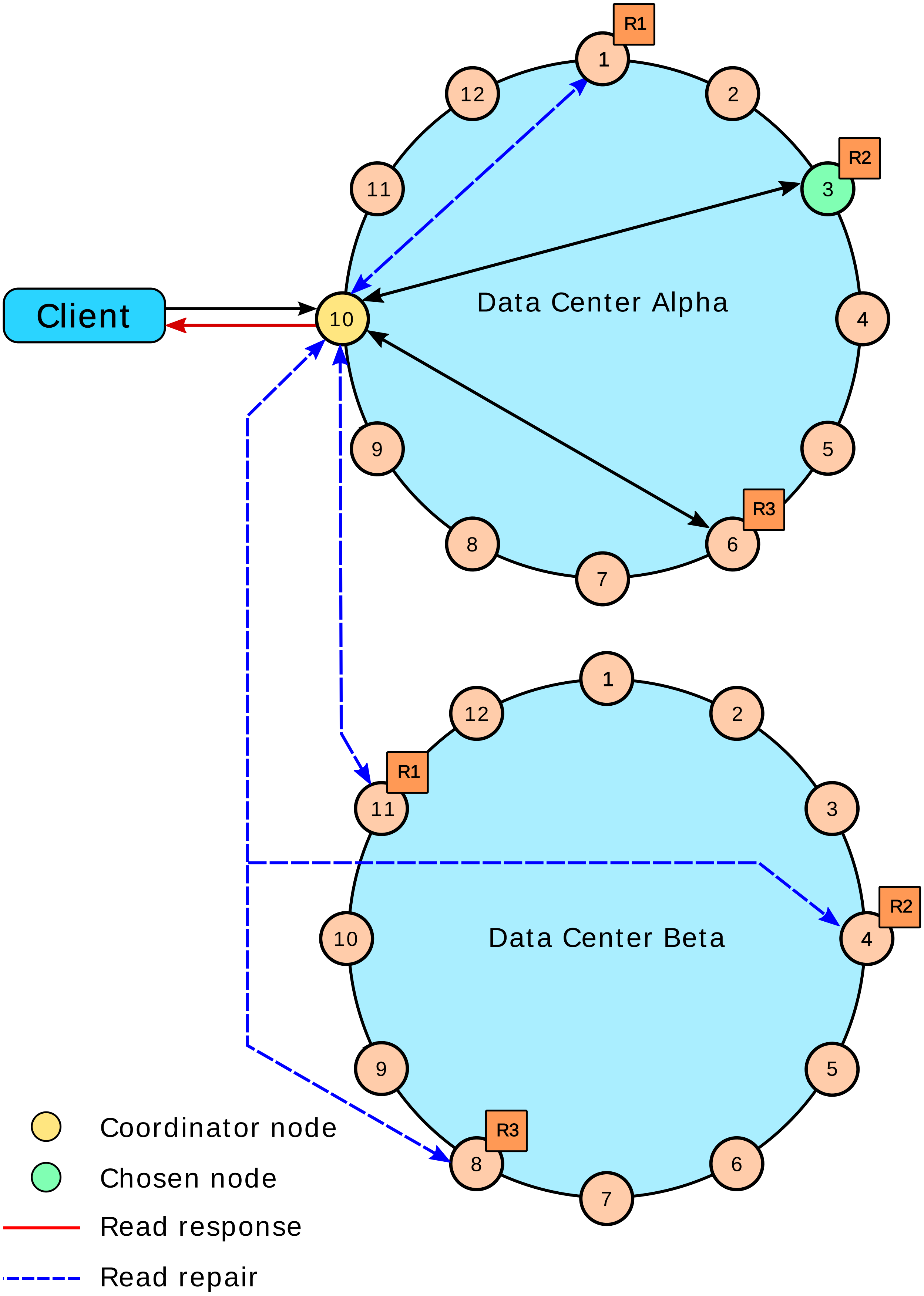
Figure: Multiple datacenter cluster with 3 replica nodes and consistency level set to QUORUM


R1
1
12
2
R2
3
11
Client
10
Data Center Alpha
4
9
5
R3
8
7
6
1
12
2
R1
11
3
R2
10
Data Center Beta
4
9
5
R3
8
7
6
Coordinator node
Chosen node
Read response
Read repair

A two datacenter cluster with a consistency level of LOCAL_QUORUM

In a multiple datacenter cluster with a replication factor of 3, and a read consistency of LOCAL_QUORUM, 2 replicas in the same datacenter as the coordinator node for the given row must respond to fulfill the read request. In the background, the remaining replicas are checked for consistency with the first 2, and if needed, a read repair is initiated for the out-of-date replicas.

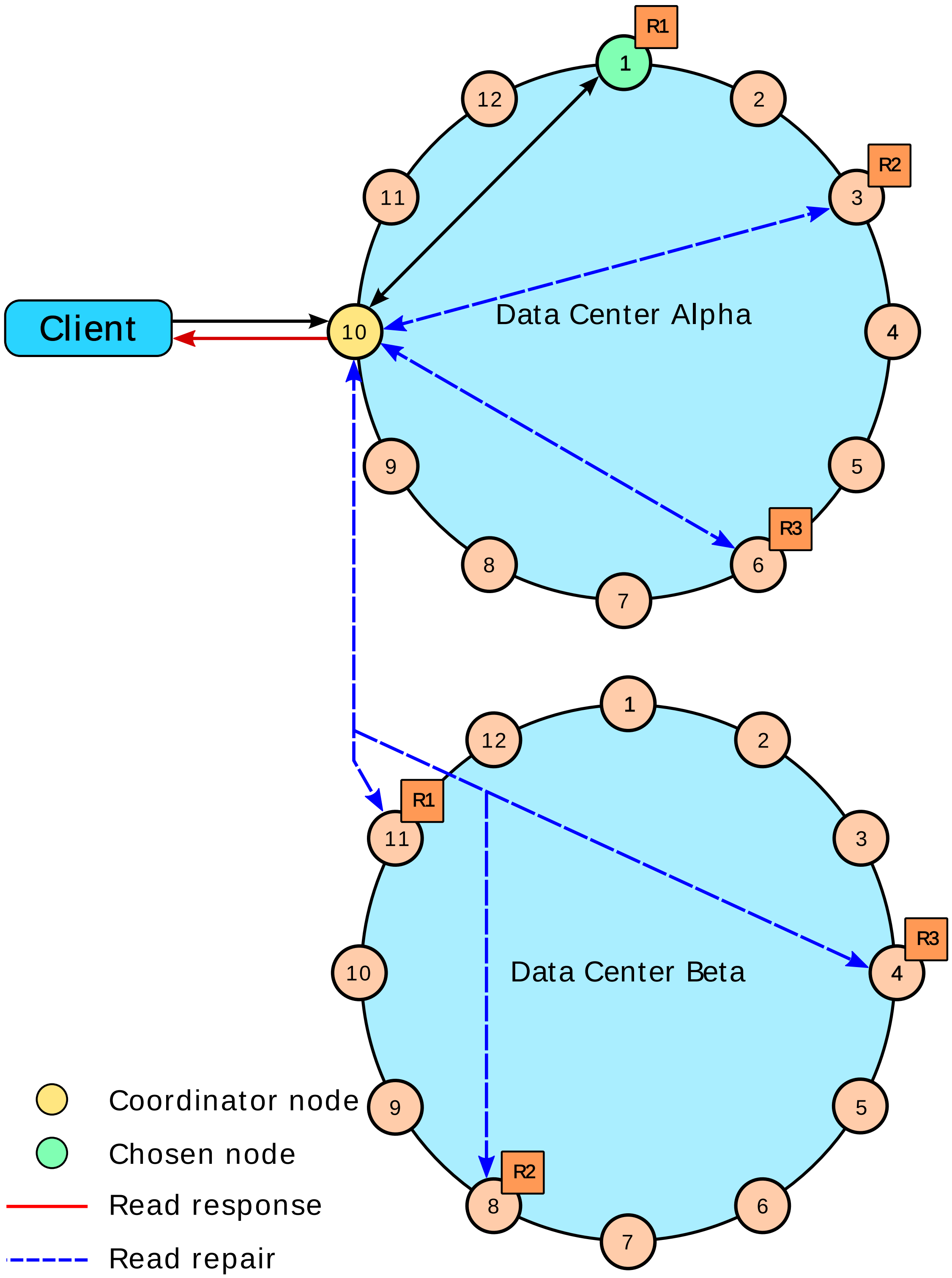
Figure: Multiple datacenter cluster with 3 replica nodes and consistency set to LOCAL_QUORUM


Client
10
Data Center Alpha
1
2
3
4
5
6
7
8
9
11
12
R1
R2
R3
Data Center Beta
1
2
3
4
5
6
7
8
9
10
11
12
R1
R2
R3
Coordinator node
Chosen node
Read response
Read repair

A two datacenter cluster with a consistency level of ONE

In a multiple datacenter cluster with a replication factor of 3, and a read consistency of ONE, the closest replica for the given row, regardless of datacenter, is contacted to fulfill the read request. In the background a read repair is potentially initiated, based on the read_repair_chance setting of the table, for the other replicas.

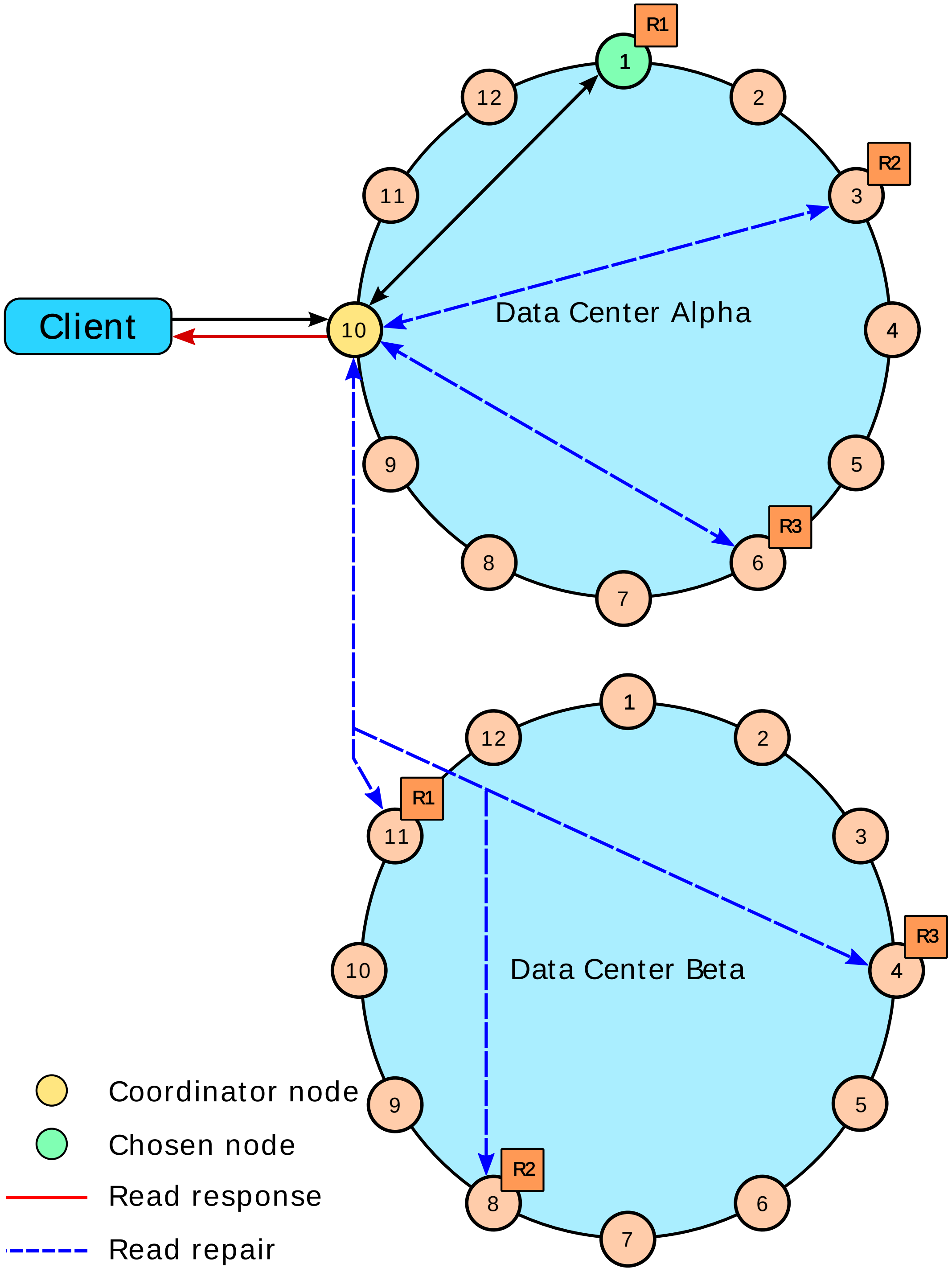
Figure: Multiple datacenter cluster with 3 replica nodes and consistency set to ONE



A two datacenter cluster with a consistency level of `LOCAL_ONE`

In a multiple datacenter cluster with a replication factor of 3, and a read consistency of `LOCAL_ONE`, the closest replica for the given row in the same datacenter as the coordinator node is contacted to fulfill the read request. In the background a read repair is potentially initiated, based on the `read_repair_chance` setting of the table, for the other replicas.

Figure: Multiple datacenter cluster with 3 replica nodes and consistency set to LOCAL_ONE



How are write requests accomplished?

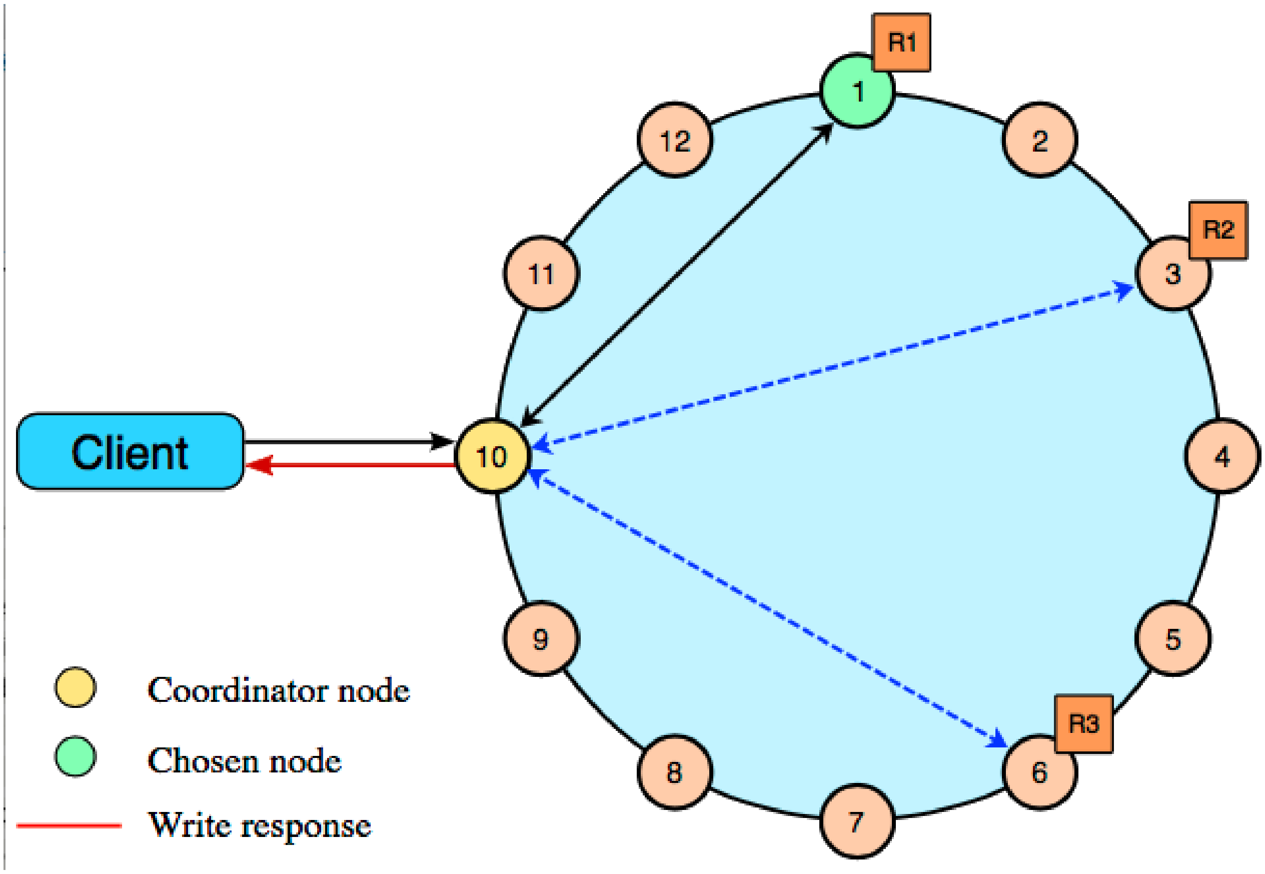
The coordinator sends a write request to *all* replicas that own the row being written. As long as all replica nodes are up and available, they will get the write regardless of the consistency level specified by the client. The write consistency level determines how many replica nodes must respond with a success acknowledgment in order for the write to be considered successful. Success means that the data was written to the commit log and the memtable as described in how data is written.

For example, in a single datacenter 10 node cluster with a replication factor of 3, an incoming write will go to all 3 nodes that own the requested row. If the write consistency level specified by the client is ONE, the first node to complete the write responds back to the coordinator, which then proxies the success message back to the client. A consistency level of ONE means that it is possible that 2 of the 3 replicas could miss the write if they happened to be down at the time the request was made. If a replica misses a write, Cassandra will make the row consistent later using one of its built-in repair mechanisms: hinted handoff, read repair, or anti-entropy node repair.

That node forwards the write to all replicas of that row. It responds to the client once it receives write acknowledgments from the number of nodes specified by the consistency level. Exceptions:

- If the coordinator cannot write to enough replicas to meet the requested Consistency level, it throws an `Unavailable` Exception and does not perform any writes.
- If there are enough replicas available but the required writes don't finish within the timeout window, the coordinator throws a `Timeout` Exception.

Figure: Single datacenter cluster with 3 replica nodes and consistency set to ONE

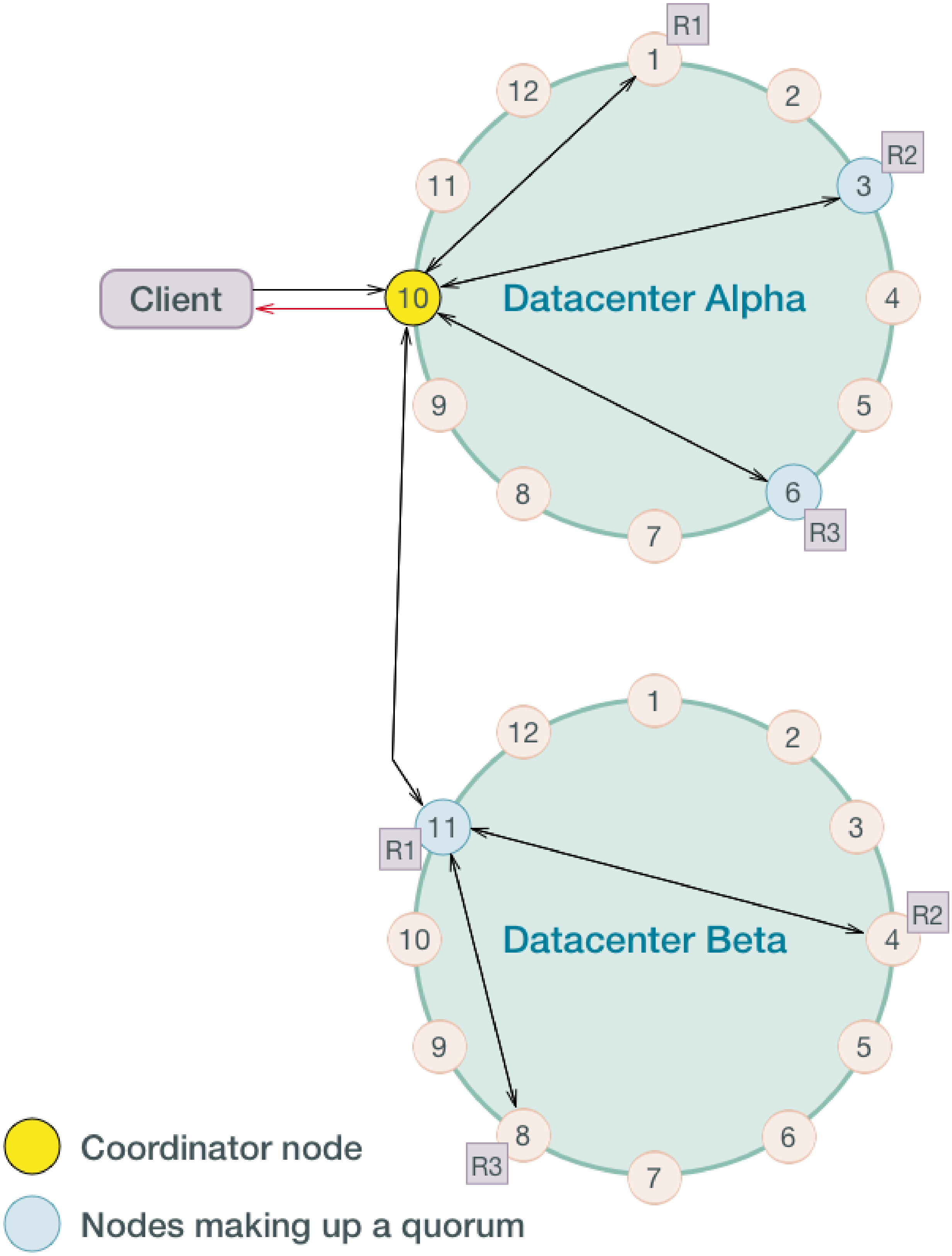


Multiple datacenter write requests

In multiple datacenter deployments, Cassandra optimizes write performance by choosing one coordinator node. The coordinator node contacted by the client application forwards the write request to one replica in each of the other DCs, with a special tag to forward the write to the other local replicas.

If the consistency level is `LOCAL_ONE` or `LOCAL_QUORUM`, only the nodes in the same datacenter as the coordinator node must respond to the client request in order for the request to succeed. This way, geographical latency does not impact client request response times.

Figure: Multiple datacenter cluster with 3 replica nodes and consistency set to QUORUM


R1
1
12
2
R2
11
3
Client
10
Datacenter Alpha
4
9
5
8
6
7
R3
1
12
2
11
3
R1
R2
10
Datacenter Beta
4
9
5
8
6
R3
7
Coordinator node
Nodes making up a quorum

Planning a cluster deployment

Planning has moved to Planning and testing cluster deployments.

Installing DataStax Distribution of Apache Cassandra 3.x

Installing the DataStax Distribution of Apache Cassandra 3.x on RHEL-based systems

Attention: OpsCenter is not supported or installed with Cassandra 2.2 and later.

Use these steps to install Cassandra using Yum repositories on RHEL, CentOS, and Oracle Linux.

Note: To install on SUSE, use the Cassandra binary tarball distribution.

Prerequisites

- Ensure that your platform is supported.
- Yum Package Management application installed.
- Root or sudo access to the install machine.
- Latest version of Oracle Java Platform, Standard Edition 8 (JDK) or OpenJDK 8.

  Note: The JDK is recommended for development and production systems. It provides tools that are not in the JRE, such as jstack, jmap, jps, and jstat, that are useful for troubleshooting.
- Python 2.7.

The packaged releases create a `cassandra` user. When starting Cassandra as a service, the service runs as this user. The following utilities are included in a separate package: sstable2json, sstablelevelreset, sstablemetadata, json2sstable, sstablerepairedset, sstablesplit, and token-generator.

Procedure

In a terminal window:

1. Check which version of Java is installed by running the following command:

   ```
   $ java -version
   ```

   It is recommended to use the latest version of Oracle Java 8 or OpenJDK 8 on all nodes.
2. Add the DataStax Distribution of Apache Cassandra 3.x repository to the `/etc/yum.repos.d/datastax.repo`:

   ```
   [datastax-ddc]
   name = DataStax Repo for Apache Cassandra
   baseurl = http://rpm.datastax.com/datastax-ddc/3.version_number
   enabled = 1
   ```

```
gpgcheck = 0
```

Note: Be sure to specify the version number. For example: 3.2.

3. Install the latest packages:

```
$ sudo yum install datastax-ddc
```

This command automatically installs the Cassandra utilities such as sstablelevelreset, sstablemetadata, sstableofflinerelevel, sstablerepairedset, sstablesplit, token-generator. Each utility provides usage/help information; type help after entering the command.

4. Optional: Single-node cluster installations only.

a) Start Cassandra:

```
$ sudo service cassandra start
```

On some Linux distributions, you many need to use:

```
$ sudo /etc/init.d/cassandra start
```

Note: Cassandra 3.8 and later: Startup is aborted if corrupted transaction log files are found and the affected log files are logged. See the log files for information on resolving the situation.

b) Verify that DataStax Distribution of Apache Cassandra is running:

```
$ nodetool status
```

```
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address            Load       Tokens  Owns    Host ID
                 Rack
UN  127.0.0.147.66 KB   47.66 KB   256     100%    aaa1b7c1-6049-4a08-
ad3e-3697a0e30e10  rack1
```

What to do next

- Configure DataStax Community
- Initializing a multiple node cluster (single datacenter) on page 132
- Initializing a multiple node cluster (multiple datacenters) on page 135
- Recommended production settings
- Key components for configuring Cassandra
- Starting Cassandra as a service on page 138
- Package installation directories on page 72

Installing DataStax Distribution of Apache Cassandra 3.x on Debian-based systems

Attention: OpsCenter is not supported or installed with Cassandra 2.2 and later.

Use these steps to install Cassandra using APT repositories on Debian and Ubuntu Linux.

Prerequisites

- Ensure that your platform is supported.

- Advanced Package Tool is installed.
- Root or sudo access to the install machine.
- Latest version of Oracle Java Platform, Standard Edition 8 (JDK) or OpenJDK 8.

  Note: The JDK is recommended for development and production systems. It provides tools that are not in the JRE, such as jstack, jmap, jps, and jstat, that are useful for troubleshooting.
- Python 2.7.

The packaged releases create a `cassandra` user. When starting Cassandra as a service, the service runs as this user. The following utilities are included in a separate package: sstable2json, sstablelevelreset, sstablemetadata, json2sstable, sstablerepairedset, sstablesplit, and token-generator.

Procedure

In a terminal window:

1. Check which version of Java is installed by running the following command:

   ```
   $ java -version
   ```

   It is recommended to use the latest version of Oracle Java 8 or OpenJDK 8 on all nodes.
2. Add the DataStax Distribution of Apache Cassandra 3.x repository to the `/etc/apt/sources.list.d/cassandra.sources.list`

   ```
   $ echo "deb http://debian.datastax.com/datastax-ddc 3.version_number main" |
    sudo tee -a /etc/apt/sources.list.d/cassandra.sources.list
   ```

   Note: Be sure to specify the version number. For example: 3.2.
3. Optional: On Debian systems, to allow installation of the Oracle JVM instead of the OpenJDK JVM:

   a) In `/etc/apt/sources.list`, find the line that describes your source repository for Debian and add `contrib non-free` to the end of the line. For example:

   ```
   deb http://some.debian.mirror/debian/ $distro main contrib non-free
   ```

   b) Save and close the file when you are done.
4. Add the DataStax repository key to your aptitude trusted keys.

   ```
   $ curl -L https://debian.datastax.com/debian/repo_key | sudo apt-key add -
   ```
5. Install the latest package:

   ```
   $ sudo apt-get update
   $ sudo apt-get install datastax-ddc
   ```

   This command automatically installs the Cassandra utilities such as sstablelevelreset, sstablemetadata, sstableofflinerelevel, sstablerepairedset, sstablesplit, token-generator. Each utility provides usage/help information; type help after entering the command.
6. Optional: Single-node cluster installations only.

   a) Because the Debian packages start the Cassandra service automatically, you do not need to start the service.

   Note: Cassandra 3.8 and later: Startup is aborted if corrupted transaction log files are found and the affected log files are logged. See the log files for information on resolving the situation.

   b) Verify that DataStax Distribution of Apache Cassandra is running:

   ```
   $ nodetool status
   ```

   ```
   Datacenter: datacenter1
   ```

```
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address              Load       Tokens   Owns     Host ID
                Rack
UN  127.0.0.147.66 KB   47.66 KB   256      100%     aaa1b7c1-6049-4a08-
ad3e-3697a0e30e10  rack1
```

7. Because the Debian packages start the Cassandra service automatically, you must stop the server and clear the data:

   Doing this removes the default cluster_name (Test Cluster) from the system table. All nodes must use the same cluster name.

   ```
   $ sudo service cassandra stop
   $ sudo rm -rf /var/lib/cassandra/data/system/*
   ```

What to do next

- Configure DataStax Community
- Initializing a multiple node cluster (single datacenter) on page 132
- Initializing a multiple node cluster (multiple datacenters) on page 135
- Recommended production settings
- Key components for configuring Cassandra
- Starting Cassandra as a service on page 138
- Package installation directories on page 72

Related tasks

Starting Cassandra as a service on page 138

Related reference

Package installation directories on page 72

Installing DataStax Distribution of Apache Cassandra 3.x on any Linux-based platform

Attention: OpsCenter is not supported or installed with Cassandra 2.2 and later.

Use these steps to install Cassandra on all Linux-based platforms using a binary tarball.

Use this install for Mac OS X and platforms without package support, or if you do not have or want a root installation.

Prerequisites

- Ensure that your platform is supported.
- Latest version of Oracle Java Platform, Standard Edition 8 (JDK) or OpenJDK 8.

  Note: The JDK is recommended for development and production systems. It provides tools that are not in the JRE, such as jstack, jmap, jps, and jstat, that are useful for troubleshooting.
- Python 2.7.

The binary tarball runs as a stand-alone process.

Procedure

In a terminal window:

1. Check which version of Java is installed by running the following command:

   ```
   $ java -version
   ```

   It is recommended to use the latest version of Oracle Java 8 or OpenJDK 8 on all nodes.
2. Download the DataStax Distribution of Apache Cassandra 3.x:

   ```
   $ curl -L http://downloads.datastax.com/datastax-ddc/datastax-
   ddc-version_number-bin.tar.gz | tar xz
   ```

   To view the available versions, see the DataStax Distribution of Apache Cassandra download page. If you download from this page, use the following command to untar:

   ```
   $ tar -xvzf datastax-ddc-version_number-bin.tar.gz
   ```

   Note: Cassandra utilities, such as sstablelevelreset, sstablemetadata, sstableofflinerelevel, sstablerepairedset, sstablesplit, and token-generator, are automatically installed. Each utility provides usage/help information; type help after entering the command.
3. For instructions about configuring Cassandra for use without root permissions, click here.
4. To configure Cassandra, go to the `install/conf` directory:

   ```
   $ cd datastax-ddc-version_number/conf
   ```
5. Optional: Single-node cluster installations only.
   a) Start Cassandra:

      ```
      $ cd install_location
      $ bin/cassandra ## use -f to start Cassandra in the foreground
      ```

      Note: Cassandra 3.8 and later: Startup is aborted if corrupted transaction log files are found and the affected log files are logged. See the log files for information on resolving the situation.

   b) Verify that DataStax Distribution of Apache Cassandra is running:

      ```
      $ install_location
      /bin/nodetool status
      ```

      ```
      Datacenter: datacenter1
      =======================
      Status=Up/Down
      |/ State=Normal/Leaving/Joining/Moving
      --  Address             Load       Tokens   Owns    Host ID
                       Rack
      UN  127.0.0.147.66 KB   47.66 KB   256      100%    aaa1b7c1-6049-4a08-
      ad3e-3697a0e30e10  rack1
      ```

What to do next

- Key components for configuring Cassandra
- Tarball installation directories on page 71
- Starting Cassandra as a stand-alone process on page 138

Configuring Cassandra without root permissions

Before performing this steps, you must have completed steps 1 and 2 in Installing from the binary tarball.

Procedure

1. In the install directory, create the data and log directories:

   ```
   $ mkdir cassandra-data; cd cassandra-data
   $ mkdir data saved_caches commitlog
   ```
2. Edit the `cassandra.yaml` file:
   a) `cd path_to_install/conf/`
   b) Edit these settings:

   ```
   data_file_directories: path_to_install/cassandra-data/data
   commitlog_directory: path_to_install/cassandra-data/commitlog
   saved_caches_directory: path_to_install/cassandra-data/saved_caches
   ```
3. Go back to the tarball installation instructions.

Installing earlier releases of DataStax Distribution of Apache Cassandra 3.x

To install the same version as other nodes in your cluster, see:

- Installing earlier packages on RHEL-based platforms
- Installing earlier packages on Debian-based platforms
- Earlier package installs on binary tarball

Installing earlier packages on RHEL-based platforms

Follow the install instructions in Using the Yum repository and specify the version in the `/etc/yum.repos.d/datastax.repo` file.

Installing earlier packages on Debian-based platforms

Follow the install instructions in Using the APT repository and specify the *version_number* in the install command.

Installing using the binary tarball

Cassandra 3.2 examples:

- Using curl:

  ```
  $ curl -L  http://downloads.datastax.com/datastax-ddc/datastax-ddc-3.2.0-
  bin.tar.gz | tar xz
  ```
- Downloading the tarball using the URL for the prior version and then extracting:
  1. Download:

     ```
     http://downloads.datastax.com/datastax-ddc/datastax-ddc-3.2.0-bin.tar.gz
     ```
  2. Unpack the distribution. For example:

     ```
     $ tar -xzvf datastax-ddc-3.2.0-bin.tar.gz
     ```

     The files are extracted into the ddc-cassandra-3.2.0 directory.

Uninstalling DataStax Distribution of Apache Cassandra 3.x

Select the uninstall method for your type of installation:

- Debian- and RHEL-based packages
- Binary tarball

Uninstalling Debian- and RHEL-based packages

Use this method when you have installed Cassandra using APT or Yum.

1. Stop the Cassandra service:

```
$ sudo service cassandra stop
```

2. Make sure all services are stopped:

```
$ ps auwx | grep cassandra
```

3. If services are still running, use the PID to kill the service:

```
$ sudo kill cassandra_pid
```

4. Remove the library and log directories:

```
$ sudo rm -r /var/lib/cassandra
$ sudo rm -r /var/log/cassandra
```

5. Remove the installation directories:

   RHEL-based packages:

```
$ sudo yum remove "cassandra-*"
```

   Debian-based packages:

```
$ sudo apt-get purge "cassandra-*"
```

Uninstalling the binary tarball

Use this method when you have installed Apache Cassandra 3.0 using the binary tarball.

1. Stop the node:

```
$ ps auwx | grep cassandra
$ sudo  kill cassandra_pid
```

2. Remove the installation directory.

Installing on cloud providers

You can install Apache Cassandra on cloud providers that supply any of the supported platforms.

You can install Cassandra 2.1 and earlier versions on Amazon EC2 using the DataStax AMI (Amazon Machine Image) as described in the AMI documentation for Cassandra 2.1.

Note: This project is currently in a maintenance mode until December 2016. During this timeframe, DataStax will no longer provide updates for DataStax ComboAMI. After this date, DataStax will stop hosting the central service and delete this repository. For more information, see the readme for this project.

To install Cassandra 2.2 and later on Amazon EC2, use a trusted AMI for your platform and the appropriate install method for that platform.

Installing the JDK

Installing Oracle JDK on RHEL-based Systems

Configure your operating system to use the latest version of Oracle Java Platform, Standard Edition 8.

Procedure

1. Check which version of the JDK your system is using:

   ```
   $ java -version
   ```

   If Oracle Java is used, the results should look like:

   ```
   java version "1.8.0_65"
   Java(TM) SE Runtime Environment (build 1.8.0_65-b17)
   Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)
   ```
2. If necessary, go to Oracle Java SE Downloads, accept the license agreement, and download the installer for your distribution.

   Note: If installing the Oracle JDK in a cloud environment, accept the license agreement, download the installer to your local client, and then use `scp` (secure copy) to transfer the file to your cloud machines.
3. From the directory where you downloaded the package, run the install:

   ```
   $ sudo rpm -ivh jdk-8uversion-linux-x64.rpm
   ```

   The RPM installs the JDK into the `/usr/java/` directory.
4. Set your system to use the Oracle JDK:

   ```
   $ sudo alternatives --install /usr/bin/java java /usr/java/jdk1.8.0_version/
   bin/java 200000
   ```
5. Use the `alternatives` command to switch to the Oracle JDK.

   ```
   $ sudo alternatives --config jav
   ```

   Note: If you have trouble, you may need to set JAVA_HOME and PATH in your profile, such as `.bash_profile`.

   The following examples assume that the JDK is in `/usr/java` and `which java` shows `/usr/bin/java`:

   - Shell or bash:

     ```
     $ export JAVA_HOME=/usr/java/latest
     $ export PATH=$JAVA_HOME/bin:$PATH
     ```
   - C shell (csh):

     ```
     $ setenv JAVA_HOME "/usr/java/latest"
     $ setenv PATH $JAVA_HOME/bin:$PATH
     ```

6. Make sure your system is using the correct JDK:

   ```
   $ java -version
   ```

   ```
   java version "1.8.0_65"
   Java(TM) SE Runtime Environment (build 1.8.0_65-b17)
   Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)
   ```

Installing Oracle JDK on Debian or Ubuntu Systems

Configure your operating system to use the latest version of Oracle Java Platform, Standard Edition 8.

The Oracle Java Platform, Standard Edition (JDK) has been removed from the official software repositories of Ubuntu and only provides a binary (`.bin`) version. You can get the JDK from the Java SE Downloads.

Procedure

1. Check which version of the JDK your system is using:

   ```
   $ java -version
   ```

   If Oracle Java is used, the results should look like:

   ```
   java version "1.8.0_65"
   Java(TM) SE Runtime Environment (build 1.8.0_65-b17)
   Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)
   ```

2. If necessary, go to Oracle Java SE Downloads, accept the license agreement, and download the installer for your distribution.

   Note: If installing the Oracle JDK in a cloud environment, accept the license agreement, download the installer to your local client, and then use `scp` (secure copy) to transfer the file to your cloud machines.

3. Make a directory for the JDK:

   ```
   $ sudo mkdir -p /usr/lib/jvm
   ```

4. Unpack the tarball and install the JDK:

   ```
   $ sudo tar zxvf jdk-8u65-linux-x64.tar.gz -C /usr/lib/jvm
   ```

   The JDK files are installed into a directory called `/usr/lib/jvm/jdk-8u_version`.

5. Tell the system that there's a new Java version available:

   ```
   $ sudo update-alternatives --install "/usr/bin/java" "java" "/usr/lib/jvm/
   jdk1.8.0_version/bin/java" 1
   ```

   If updating from a previous version that was removed manually, you many need to execute the above command twice, because you'll get an error message the first time.

6. Set the new JDK as the default using the following command:

   ```
   $ sudo update-alternatives --config java
   ```

7. Make sure your system is using the correct JDK:

   ```
   $ java -version
   ```

   ```
   java version "1.8.0_65"
   ```

```
Java(TM) SE Runtime Environment (build 1.8.0_65-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)
```

Installing OpenJDK on RHEL-based Systems

Configure your operating system to use the OpenJDK 8.

Procedure

In a terminal:

1. Install the OpenJDK 8:

```
$ su -c "yum install java-1.8.0-openjdk"
```

2. If you have more than one Java version installed on your system use the following command to switch versions:

```
$ sudo alternatives --config java
```

3. Make sure your system is using the correct JDK:

```
$ java -version
```

```
openjdk version "1.8.0_71"
OpenJDK Runtime Environment (build 1.8.0_71-b15)
OpenJDK 64-Bit Server VM (build 25.71-b15, mixed mode)
```

Installing OpenJDK on Debian-based Systems

Configure your operating system to use the OpenJDK 8.

Procedure

In a terminal:

1. Install the OpenJDK 8 from a PPA repository:

```
$ sudo add-apt-repository ppa:openjdk-r/ppa
```

2. Update the system package cache and install:

```
$ sudo apt-get update
$ sudo apt-get install openjdk-8-jdk
```

3. If you have more than one Java version installed on your system use the following command to switch versions:

```
$ sudo update-alternatives --config java
```

4. Make sure your system is using the correct JDK:

```
$ java -version
```

```
openjdk version "1.8.0_72-internal"
OpenJDK Runtime Environment (build 1.8.0_72-internal-b05)
OpenJDK 64-Bit Server VM (build 25.72-b05, mixed mode)
```

Recommended production settings for Linux and Windows

Recommendations for production environments have been moved to Recommended production settings.

Install locations

Tarball installation directories

The configuration files are located in the following directories:

Configuration and sample files	Locations	Description
cassandra.yaml	*install_location*/conf	Main configuration file.
cassandra-env.sh	*install_location*/conf	Linux settings for Java, some JVM, and JMX.
jvm.options	*install_location*/conf	Static JVM settings for heap, garbage collection, and Cassandra startup parameters.
cassandra.in.sh	*install_location*/bin	Sets environment variables.
cassandra-rackdc.properties	*install_location*/conf	Defines the default datacenter and rack used by the GossipingPropertyFileSnitch, Ec2Snitch, Ec2MultiRegionSnitch, and GoogleCloudSnitch.
cassandra-topology.properties	*install_location*/conf	Defines the default datacenter and rack used by the PropertyFileSnitch.
commit_archiving.properties	*install_location*/conf	Configures commitlog archiving.
cqlshrc.sample	*install_location*/conf	Example file for using cqlsh with SSL encryption.
metrics-reporter-config-sample.yaml	*install_location*/conf	Example file for configuring metrics in Cassandra.
logback.xmlcat	*install_location*/conf	Configuration file for logback.
triggers	*install_location*/conf	The default location for the trigger JARs.

The binary tarball releases install into the installation directory.

Directories	Description
bin	Utilities and start scripts.
conf	Configuration files and environment settings.

Directories	Description
`data`	Directory containing the files for commitlog, data, and saved_caches (unless set in `cassandra.yaml`.)
`interface`	Thrift legacy API.
`javadoc`	Cassandra Java API documentation.
`lib`	JAR and license files.
`tools`	Cassandra tools and sample `cassandra.yaml` files for stress testing.

For DataStax Enterprise installs, see the documentation for your DataStax Enterprise version.

Package installation directories

The configuration files are located in the following directories:

Configuration Files	Locations	Description
`cassandra.yaml`	`/etc/cassandra`	Main configuration file.
`cassandra-env.sh`	`/etc/cassandra`	Linux settings for Java, some JVM, and JMX.
`jvm.options`	`/etc/cassandra`	Static JVM settings for heap, garbage collection, and Cassandra startup parameters.
`cassandra.in.sh`	`/usr/share/cassandra`	Sets environment variables.
`cassandra-rackdc.properties`	`/etc/cassandra`	Defines the default datacenter and rack used by the GossipingPropertyFileSnitch, Ec2Snitch, Ec2MultiRegionSnitch, and GoogleCloudSnitch.
`cassandra-topology.properties`	`/etc/cassandra`	Defines the default datacenter and rack used by the PropertyFileSnitch.
`commit_archiving.properties`	`/etc/cassandra`	Configures commitlog archiving.
`cqlshrc.sample`	`/etc/cassandra`	Example file for using cqlsh with SSL encryption.
`logback.xml`	`/etc/cassandra`	Configuration file for logback.
`triggers`	`/etc/cassandra`	The default location for the trigger JARs.

The packaged releases install into these directories:

Directories	Description
`/etc/default`	
`/etc/init.d/cassandra`	Service startup script.
`/etc/security/limits.d`	Cassandra user limits.
`/etc/cassandra`	Configuration files.
`/usr/bin`	Binary files.
`/usr/sbin`	

Directories	Description
`/usr/share/doc/cassandra/examples`	Sample `yaml` files for stress testing.
`/usr/share/cassandra`	JAR files and environment settings (`cassandra.in.sh`).
`/usr/share/cassandra/lib`	JAR files.
`/var/lib/cassandra`	Data, commitlog, and saved_caches directories.
`/var/log/cassandra`	Log directory.
`/var/run/cassandra`	Runtime files.

For DataStax Enterprise installs, see the documentation for your DataStax Enterprise version.

Configuration

The cassandra.yaml configuration file

The `cassandra.yaml` file is the main configuration file for Cassandra.

Important: After changing properties in the `cassandra.yaml` file, you must restart the node for the changes to take effect. It is located in the following directories:

- Cassandra package installations: `/etc/cassandra`
- Cassandra tarball installations: ***`install_location`***`/conf`

The configuration properties are grouped into the following sections:

- Quick start

  The minimal properties needed for configuring a cluster.
- Commonly used

  Properties most frequently used when configuring Cassandra.
- Performance tuning

  Tuning performance and system resource utilization, including commit log, compaction, memory, disk I/O, CPU, reads, and writes.
- Advanced

  Properties for advanced users or properties that are less commonly used.
- Security

  Server and client security settings.

Note: Values with $^{\textit{note}}$ mark default values that are defined internally, missing, or commented out, or whose implementation depends on other properties in the `cassandra.yaml` file. Additionally, some commented-out values may not match the actual default values. These are recommended alternatives to the default values.

Quick start properties

The minimal properties needed for configuring a cluster.

Related information: Initializing a multiple node cluster (single datacenter) on page 132 and Initializing a multiple node cluster (multiple datacenters) on page 135.

cluster_name

(Default: Test Cluster) The name of the cluster. This setting prevents nodes in one logical cluster from joining another. All nodes in a cluster must have the same value.

listen_address

(Default: localhost) The IP address or hostname that Cassandra binds to for connecting this node to other nodes. Set this parameter or listen_interface, not both. Correct settings for various use cases:

- **Single-node installations**: do one of the following:
  - Comment this property out. If the node is properly configured (host name, name resolution, and so on.), Cassandra uses `InetAddress.getLocalHost()` to get the local address from the system.
  - Leave set to the default, `localhost`.
- **Node in a multi-node installations**: set this property to the node's IP address or hostname, or set listen_interface.
- **Node in a multi-network or multi-Datacenter installation, within an EC2 environment that supports automatic switching between public and private interfaces**: set `listen_address` to the node's IP address or hostname, or set listen_interface.
- **Node with two physical network interfaces in a multi-datacenter installation or a Cassandra cluster deployed across multiple Amazon EC2 regions using the `Ec2MultiRegionSnitch`**:
  1. Set `listen_address` to this node's private IP or hostname, or set listen_interface (for communication within the local datacenter).
  2. Set broadcast_address to the second IP or hostname (for communication between datacenters).
  3. Set listen_on_broadcast_address to `true`.
  4. If this node is a seed node, add the node's *public* IP address or hostname to the seeds list.
- Open the storage_port or ssl_storage_port on the public IP firewall.

Warning:

- Never set `listen_address` to 0.0.0.0. It is always wrong.
- Do not set values for both listen_address and `listen_interface` on the same node.

listen_interface

(Default: eth0)[note] The interface that Cassandra binds to for connecting to other Cassandra nodes. Interfaces must correspond to a single address — IP aliasing is not supported. Do not set values for both listen_address and `listen_interface` on the same node.

listen_interface_prefer_ipv6

(Default: false) If an interface has an ipv4 and an ipv6 address, Cassandra uses the first ipv4 address by default. Set this property to true to configure Cassandra to use the first ipv6 address.

Default directories

If you have changed any of the default directories during installation, set these properties to the new locations. Make sure you have root access.

commitlog_directory

The directory where the commit log is stored. Default locations:

For optimal write performance, place the commit log be on a separate disk partition, or (ideally) a separate physical device from the data file directories. Because the commit log is append only, an HDD is acceptable for this purpose.

data_file_directories

The directory location where table data is stored (in SSTables). Cassandra distributes data evenly across the location, subject to the granularity of the configured compaction strategy. Default locations:

As a production best practice, use RAID 0 and SSDs.

saved_caches_directory

The directory location where table key and row caches are stored. Default location:

- Package installations: `/var/lib/cassandra/saved_caches`
- Tarball installations: *`install_location`*`/data/saved_caches`

Commonly used properties

Properties most frequently used when configuring Cassandra.

Before starting a node for the first time, you should carefully evaluate your requirements.

Common initialization properties

Note: Be sure to set the properties in the Quick start section as well.

commit_failure_policy

(Default: `stop`) Policy for commit disk failures:

- die

  Shut down gossip and Thrift and kill the JVM, so the node can be replaced.
- stop

  Shut down gossip and Thrift, leaving the node effectively dead, available for inspection using JMX.
- stop_commit

  Shut down the commit log, letting writes collect but continuing to service reads (as in pre-2.0.5 Cassandra).
- ignore

  Ignore fatal errors and let the batches fail.

disk_optimization_strategy

(Default: ssd) The strategy for optimizing disk reads. Possible values: ssd or spinning.

disk_failure_policy

(Default: stop) Sets how Cassandra responds to disk failure. Recommend settings: stop or best_effort. Valid values:

- die

  Shut down gossip and Thrift and kill the JVM for any file system errors or single SSTable errors, so the node can be replaced.
- stop_paranoid

  Shut down gossip and Thrift even for single SSTable errors.
- stop

  Shut down gossip and Thrift, leaving the node effectively dead, but available for inspection using JMX.
- best_effort

  Stop using the failed disk and respond to requests based on the remaining available SSTables. This allows obsolete data at consistency level of ONE.

- ignore

  Ignore fatal errors and lets the requests fail; all file system errors are logged but otherwise ignored. Cassandra acts as in versions prior to 1.2.

Related information: Handling Disk Failures In Cassandra 1.2 blog and Recovering from a single disk failure using JBOD on page 156.

endpoint_snitch

(Default: `org.apache.cassandra.locator.SimpleSnitch`) Set to a class that implements the `IEndpointSnitch` interface. Cassandra uses the snitch to locate nodes and route requests.

- **SimpleSnitch**

  Use for single-datacenter deployment or single-zone deployment in public clouds. Does not recognize datacenter or rack information. Treats strategy order as proximity, which can improve cache locality when you disable read repair.
- **GossipingPropertyFileSnitch**

  Recommended for production. Reads rack and datacenter for the local node in cassandra-rackdc.properties file and propagates these values to other nodes via gossip. For migration from the PropertyFileSnitch, uses the cassandra-topology.properties file if it is present.
- **PropertyFileSnitch**

  Determines proximity by rack and datacenter, which are explicitly configured in cassandra-topology.properties file.
- **Ec2Snitch**

  For EC2 deployments in a single region. Loads region and availability zone information from the Amazon EC2 API. The region is treated as the datacenter and the availability zone as the rack and uses only private IP addresses. For this reason, it does not work across multiple regions.
- **Ec2MultiRegionSnitch**

  Uses the public IP as the broadcast_address to allow cross-region connectivity. This means you must also set seed addresses to the public IP and open the storage_port or ssl_storage_port on the public IP firewall. For intra-region traffic, Cassandra switches to the private IP after establishing a connection.
- **RackInferringSnitch**:

  Proximity is determined by rack and datacenter, which are assumed to correspond to the 3rd and 2nd octet of each node's IP address, respectively. Best used as an example for writing a custom snitch class (unless this happens to match your deployment conventions).
- **GoogleCloudSnitch**:

  Use for Cassandra deployments on Google Cloud Platform across one or more regions. The region is treated as a datacenter and the availability zones are treated as racks within the datacenter. All communication occurs over private IP addresses within the same logical network.
- **CloudstackSnitch**

  Use the CloudstackSnitch for Apache Cloudstack environments.

Related information: Snitches on page 20

rpc_address

(Default: localhost) The listen address for client connections (Thrift RPC service and native transport). Valid values:

- unset:

  Resolves the address using the configured hostname configuration of the node. If left unset, the hostname resolves to the IP address of this node using `/etc/hostname`, `/etc/hosts`, or DNS.
- 0.0.0.0:

Listens on all configured interfaces. You must set the broadcast_rpc_address to a value other than 0.0.0.0.

- IP address
- hostname

Related information: Network

rpc_interface

(Default: eth1)[note] The listen address for client connections. Interface must correspond to a single address, IP aliasing is not supported. See rpc_address.

rpc_interface_prefer_ipv6

(Default: false) If an interface has an ipv4 and an ipv6 address, Cassandra uses the first ipv4 address by default, i. If set to true, the first ipv6 address will be used.

seed_provider

The addresses of hosts designated as contact points in the cluster. A joining node contacts one of the nodes in the -seeds list to learn the topology of the ring.

- class_name (Default: `org.apache.cassandra.locator.SimpleSeedProvider`)

  The class within Cassandra that handles the seed logic. It can be customized, but this is typically not required.
- - seeds (Default: 127.0.0.1)

  A comma-delimited list of IP addresses used by gossip for bootstrapping new nodes joining a cluster. If your cluster includes multiple nodes, you must change the list from the default value to the IP address of one of the nodes.

  Attention: In multiple data-center clusters, include at least one node from each datacenter (replication group) in the seed list. Designating more than a single seed node per datacenter is recommended for fault tolerance. Otherwise, gossip has to communicate with another datacenter when bootstrapping a node.

  Making every node a seed node is **not** recommended because of increased maintenance and reduced gossip performance. Gossip optimization is not critical, but it is recommended to use a small seed list (approximately three nodes per datacenter).

Related information: Initializing a multiple node cluster (single datacenter) on page 132 and Initializing a multiple node cluster (multiple datacenters) on page 135.

enable_user_defined_functions

(Default: false) User defined functions (UDFs) present a security risk, since they are executed on the server side. In Cassandra 3.0 and later, UDFs are executed in a sandbox to contain the execution of malicious code. They are disabled by default.

enable_scripted_user_defined_functions

(Default: false) Java UDFs are always enabled, if `enable_user_defined_functions` is true. Enable this option to use UDFs with `language javascript` or any custom JSR-223 provider. This option has no effect if `enable_user_defined_functions` is false.

Common compaction settings

compaction_throughput_mb_per_sec

(Default: 16) Throttles compaction to the specified Mb/second across the instance. The faster Cassandra inserts data, the faster the system must compact in order to keep the SSTable count down. The recommended value is 16 to 32 times the rate of write throughput (in Mb/second). Setting the value to 0 disables compaction throttling.

Related information: Configuring compaction on page 177

compaction_large_partition_warning_threshold_mb

(Default: 100) Cassandra logs a warning when compacting partitions larger than the set value.

Common memtable settings

memtable_heap_space_in_mb

(Default: 1/4 of heap size)note

The amount of on-heap memory allocated for memtables. Cassandra uses the total of this amount and the value of memtable_offheap_space_in_mb to set a threshold for automatic memtable flush. For details, see memtable_cleanup_threshold.

Related information: Tuning the Java heap

memtable_offheap_space_in_mb

(Default: 1/4 of heap size)note

Sets the total amount of off-heap memory allocated for memtables. Cassandra uses the total of this amount and the value of memtable_heap_space_in_mb to set a threshold for automatic memtable flush. For details, see memtable_cleanup_threshold.

Related information: Tuning the Java heap

Common disk settings

concurrent_reads

(Default: 32)note Workloads with more data than can fit in memory encounter a bottleneck in fetching data from disk during reads. Setting `concurrent_reads` to (16 × number_of_drives) allows operations to queue low enough in the stack so that the OS and drives can reorder them. The default setting applies to both logical volume managed (LVM) and RAID drives.

concurrent_writes

(Default: 32)note Writes in Cassandra are rarely I/O bound, so the ideal number of concurrent writes depends on the number of CPU cores on the node. The recommended value is 8 × number_of_cpu_cores.

concurrent_counter_writes

(Default: 32)note Counter writes read the current values before incrementing and writing them back. The recommended value is (16 × number_of_drives) .

concurrent_batchlog_writes

(Default: 32) Limit on the number of concurrent batchlog writes, similar to `concurrent_writes`.

concurrent_materialized_view_writes

(Default: 32) Limit on the number of concurrent materialized view writes. Set this to the lesser of concurrent reads or concurrent writes, because there is a read involved in each materialized view write.

Common automatic backup settings

incremental_backups

(Default: false) Backs up data updated since the last snapshot was taken. When enabled, Cassandra creates a hard link to each SSTable flushed or streamed locally in a `backups` subdirectory of the keyspace data. Removing these links is the operator's responsibility.

Related information: Enabling incremental backups on page 154

snapshot_before_compaction

(Default: false) Enables or disables taking a snapshot before each compaction. A snapshot is useful to back up data when there is a data format change. Be careful using this option: Cassandra does not clean up older snapshots automatically.

Related information: Configuring compaction on page 177

Common fault detection setting

phi_convict_threshold

(Default: 8)note Adjusts the sensitivity of the failure detector on an exponential scale. Generally this setting does not need adjusting.

Related information: Failure detection and recovery on page 14

Performance tuning properties

Tuning performance and system resource utilization, including commit log, compaction, memory, disk I/O, CPU, reads, and writes.

Commit log settings

commitlog_sync

(Default: periodic) The method that Cassandra uses to acknowledge writes in milliseconds:

- periodic: (Default: 10000 milliseconds [10 seconds])

  With commitlog_sync_period_in_ms, controls how often the commit log is synchronized to disk. Periodic syncs are acknowledged immediately.
- batch: (Default: disabled)[note]

  With commitlog_sync_batch_window_in_ms (Default: 2 ms), controls how long Cassandra waits for other writes before performing a sync. When this method is enabled, Cassandra does not acknowledge writes until they are fsynced to disk.

commitlog_segment_size_in_mb

(Default: 32MB) The size of an individual commitlog file segment. A commitlog segment may be archived, deleted, or recycled after all its data has been flushed to SSTables. This data can potentially include commitlog segments from every table in the system. The default size is usually suitable for most commitlog archiving, but if you want a finer granularity, 8 or 16 MB is reasonable.

By default, the max_mutation_size_in_kb is set to half of the `commitlog_segment_size_in_kb`.

Related information: Commit log archive configuration on page 129

max_mutation_size_in_kb

(Default: ½ of commitlog_segment_size_in_mb.

If a mutation's size exceeds this value, the mutation is rejected. Before increasing the commitlog segment size of the commitlog segments, investigate why the mutations are larger than expected. Look for underlying issues with access patterns and data model, because increasing the commitlog segment size is a limited fix.

Restriction:

If you set `max_mutation_size_in_kb` explicitly, then you must set `commitlog_segment_size_in_mb` to at least twice the size of max_mutation_size_in_kb / 1024.

For more information, see commitlog_segment_size_in_mb above.

commitlog_compression

(Default: not enabled) The compressor to use if commit log is compressed. Valid values: `LZ4`, `Snappy` or `Deflate`. If no value is set for this property, the commit log is written uncompressed.

commitlog_total_space_in_mb

(Default: 32MB for 32-bit JVMs, 8192MB for 64-bit JVMs)[note] Total space used for commit logs. If the total space used by all commit logs goes above this value, Cassandra rounds up to the next nearest segment multiple and flushes memtables to disk for the oldest commitlog segments, removing those log segments from the commit log. This reduces the amount of data to replay on start-up, and prevents infrequently-updated tables from keeping commitlog segments indefinitely. If the `commitlog_total_space_in_mb` is small, the result is more flush activity on less-active tables.

Related information: Configuring memtable thresholds on page 177

Compaction settings

Related information: Configuring compaction on page 177

concurrent_compactors

(Default: Smaller of number of disks or number of cores, with a minimum of 2 and a maximum of 8 per CPU core)note The number of concurrent compaction processes allowed to run simultaneously on a node, not including validation compactions for anti-entropy repair. Simultaneous compactions help preserve read performance in a mixed read-write workload by limiting the number of small SSTables that accumulate during a single long-running compaction. If your data directories are backed by SSDs, increase this value to the number of cores. If compaction running too slowly or too fast, adjust compaction_throughput_mb_per_sec first.

Note: Increasing concurrent compactors leads to more use of available disk space for compaction, because concurrent compactions happen in parallel, especially for STCS. Ensure that adequate disk space is available before increasing this configuration.

sstable_preemptive_open_interval_in_mb

(Default: 50MB) The compaction process opens SSTables before they are completely written and uses them in place of the prior SSTables for any range previously written. This setting helps to smoothly transfer reads between the SSTables by reducing page cache churn and keeps hot rows hot.

Memtable settings

memtable_allocation_type

(Default: heap_buffers) The method Cassandra uses to allocate and manage memtable memory. See Off-heap memtables in Cassandra 2.1. In releases 3.2.0 and 3.2.1, the only option that works is: heap-buffers (On heap NIO (non-blocking I/O) buffers).

memtable_cleanup_threshold

(Default: 1/(memtable_flush_writers + 1))note. Ratio used for automatic memtable flush. Cassandra adds memtable_heap_space_in_mb to memtable_offheap_space_in_mb and multiplies the total by memtable_cleanup_threshold to get a space amount in MB. When the total amount of memory used by all non-flushing memtables exceeds this amount, Cassandra flushes the largest memtable to disk.

For example, consider a node where the total of memtable_heap_space_in_mb and memtable_offheap_space_in_mb is 1000, and memtable_cleanup_threshold is `0.50`. The *memtable_cleanup* amount is `500`MB. This node has two memtables: Memtable A (150MB) and Memtable B (350MB). When either memtable increases, the total space they use exceeds 500MB and Cassandra flushes the Memtable B to disk.

A larger value for memtable_cleanup_threshold means larger flushes, less frequent flushes and potentially less compaction activity, but also less concurrent flush activity, which can make it difficult to keep your disks saturated under heavy write load.

This section documents the formula used to calculate the ratio based on the number of memtable_flush_writers. The default value in `cassandra.yaml` is `0.11`, which works if the node has many disks or if you set the node's memtable_flush_writers to `8`. As another example, if the node uses a single SSD, the value for memttable_cleanup_threshold computes to `0.33`, based on the minimum memtable_flush_writers value of `2`.

file_cache_size_in_mb

(Default: Smaller of 1/4 heap or 512) Total memory to use for SSTable-reading buffers.

buffer_pool_use_heap_if_exhausted

(Default: true)note Indicates whether Cassandra allocates allocate on-heap or off-heap memory when the SSTable buffer pool is exhausted (when the buffer pool has exceeded the maximum memory file_cache_size_in_mb), beyond this amount, Cassandra stops caching buffers, but allocates on request.

memtable_flush_writers

(Default: Smaller of number of disks or number of cores with a minimum of 2 and a maximum of 8)note The number of memtable flush writer threads. These threads are blocked by disk I/O, and each one holds a memtable in memory while blocked. If your data directories are backed by SSDs, increase this setting to the number of cores.

Cache and index settings

column_index_size_in_kb

(Default: 64) Granularity of the index of rows within a partition. For huge rows, decrease this setting to improve seek time. If you use key cache, be careful not to make this setting too large because key cache will be overwhelmed. If you're unsure of the size of the rows, it's best to use the default setting.

index_summary_capacity_in_mb

(Default: 5% of the heap size [empty])note Fixed memory pool size in MB for SSTable index summaries. If the memory usage of all index summaries exceeds this limit, any SSTables with low read rates shrink their index summaries to meet this limit. This is a best-effort process. In extreme conditions, Cassandra may use more than this amount of memory.

index_summary_resize_interval_in_minutes

(Default: 60 minutes) How frequently index summaries should be re-sampled. Re-sampling is done periodically to redistribute memory from the fixed-size pool to SSTables proportional their recent read rates. To disable, set to -1. This setting leaves existing index summaries at their current sampling level.

Disks settings

stream_throughput_outbound_megabits_per_sec

(Default: 200 Mbps)note Throttle for the throughput of all outbound streaming file transfers on a node. Cassandra does mostly sequential I/O when streaming data during bootstrap or repair. This can saturate the network connection and degrade client (RPC) performance.

inter_dc_stream_throughput_outbound_megabits_per_sec

(Default: unset)note Throttle for all streaming file transfers between datacenters, and for network stream traffic as configured with stream_throughput_outbound_megabits_per_sec.

trickle_fsync

(Default: false) When set to `true`, causes fsync to force the operating system to flush the dirty buffers at the set interval trickle_fsync_interval_in_kb. Enable this parameter to prevent sudden dirty buffer flushing from impacting read latencies. Recommended for use with SSDs, but not with HDDs.

trickle_fsync_interval_in_kb

(Default: 10240). The size of the fsync in kilobytes.

windows_timer_interval

(Default: 1) The default Windows kernel timer and scheduling resolution is 15.6ms for power conservation. Lowering this value on Windows can provide much tighter latency and better throughput. However, some virtualized environments may see a negative performance impact from changing this setting below the system default. The `sysinternals clockres` tool can confirm your system's default setting.

Advanced properties

Properties for advanced users or properties that are less commonly used.

Advanced initialization properties

auto_bootstrap

(Default: true) This setting has been removed from default configuration. It causes new (non-seed) nodes migrate the right data to themselves automatically. When initializing a fresh cluster *without* data, add auto_bootstrap: false.

Related information: Initializing a multiple node cluster (single datacenter) on page 132 and Initializing a multiple node cluster (multiple datacenters) on page 135.

batch_size_warn_threshold_in_kb

(Default: 5KB per batch) Causes Cassandra to log a WARN message when any batch size exceeds this value in kilobytes.

CAUTION: Increasing this threshold can lead to node instability.

batch_size_fail_threshold_in_kb

(Default: 50KB per batch) Cassandra fails any batch whose size exceeds this setting. The default value is 10X the value of `batch_size_warn_threshold_in_kb`.

broadcast_address

(Default: listen_address)note The "public" IP address this node uses to broadcast to other nodes outside the network or across regions in multiple-region EC2 deployments. If this property is commented out, the node uses the same IP address or hostname as `listen_address`. A node does not need a separate `broadcast_address` in a single-node or single-datacenter installation, or in an EC2-based network that supports automatic switching between private and public communication. It is necessary to set a separate `listen_address` and `broadcast_address` on a node with multiple physical network interfaces or other topologies where not all nodes have access to other nodes by their private IP addresses. For specific configurations, see the instructions for listen_address.

listen_on_broadcast_address

(Default: false) If this node uses multiple physical network interfaces, set a unique IP address for broadcast_address and set `listen_on_broadcast_address` to true. This enables the node to communicate on both interfaces.

Set this property to false if the node is on a network that automatically routes between public and private networks, as Amazon EC2 does.

For configuration details, see the instructions for listen_address.

initial_token

(Default: disabled) Set this property for single-node-per-token architecture, in which a node owns exactly one contiguous range in the ring space. Setting this property overrides num_tokens.

If your Cassandra installation is not using vnodes or this node's num_tokens is set it to 1 or is commented out, you should always set an `initial_token` value when setting up a production cluster for the first time, and when adding capacity. For more information, see this parameter in the Cassandra 1.1 Node and Cluster Configuration documentation.

This parameter can be used with num_tokens (vnodes) in special cases such as Restoring from a snapshot on page 154.

Note:

If you are using DataStax Enterprise, your node's setting for this property depends on the type of node and type of install. For more information, see Virtual node (vnode) configuration.

num_tokens

(Default: 256) note The number of tokens randomly assigned to this node in a cluster that uses virtual nodes (vnodes). This setting is evaluated in relation to the `num_tokens` set on other nodes in the cluster. If this node's `num_tokens` value is higher than the values on other nodes, the vnode logic assigns this node a larger proportion of data relative to other nodes. In general, if all nodes have equal hardware capability, each one should have the same `num_tokens` value . The recommended value is 256. If this property is commented out (`#num_tokens`), Cassandra uses 1 (equivalent to `#num_tokens : 1`) for legacy compatibility and assigns tokens using the initial_token property.

If this cluster is not using vnodes, comment out `num_tokens` or set `num_tokens: 1` and use initial_token. If you already have an existing cluster with one token per node and wish to migrate to vnodes, see Enabling virtual nodes on an existing production cluster.

Note:

If you are using DataStax Enterprise, your node's setting for this property depends on the type of node and type of install. For more information, see Configuring DataStax Enterprise in the DataStax Enterprise documentation.

allocate_tokens_keyspace

(Default: KEYSPACE) Enables automatic allocation of `num_tokens` tokens for this node. The allocation algorithm attempts to choose tokens in a way that optimizes replicated load over the nodes in the datacenter for the replication strategy used by the specified KEYSPACE. The load assigned to each node will near proportional to its number of vnodes.

partitioner

(Default: `org.apache.cassandra.dht.Murmur3Partitioner`) Sets the class that distributes rows (by partition key) across all nodes in the cluster. Any `IPartitioner` may be used, including your own as long as it is in the class path. For new clusters use the default partitioner.

Cassandra provides the following partitioners for backwards compatibility:

- `RandomPartitioner`
- `ByteOrderedPartitioner` (deprecated)
- `OrderPreservingPartitioner` (deprecated)

Related information: Partitioners on page 18

storage_port

(Default: 7000) The port for inter-node communication.

tracetype_query_ttl

(Default: 86400) TTL for different trace types used during logging of the query process

tracetype_repair_ttl

(Default: 604800) TTL for different trace types used during logging of the repair process.

Advanced automatic backup setting

auto_snapshot

(Default: true) Enables or disables whether Cassandra takes a snapshot of the data before truncating a keyspace or dropping a table. To prevent data loss, Datastax strongly advises using the default setting. If you set `auto_snapshot` to `false`, you lose data on truncation or drop.

Key caches and global row properties

When creating or modifying tables, you can enable or disable the key cache (partition key cache) or row cache for that table by setting the caching parameter. Other row and key cache tuning and configuration options are set at the global (node) level. Cassandra uses these settings to automatically distribute memory for each table on the node based on the overall workload and specific table usage. You can also configure the save periods for these caches globally.

Related information: Configuring caches

key_cache_keys_to_save

(Default: disabled - all keys are saved)[note] Number of keys from the key cache to save.

key_cache_save_period

(Default: 14400 seconds [4 hours]) Duration in seconds that keys are kept in cache. Caches are saved to saved_caches_directory. Saved caches greatly improve cold-start speeds and have relatively little effect on I/O.

key_cache_size_in_mb

(Default: empty) A global cache setting for the maximum size of the key cache in memory (for all tables). If no value is set, the cache is set to the smaller of 5% of the available heap, or 100MB. To disable set to 0.

Related information: setcachecapacity, Enabling and configuring caching on page 175.

column_index_cache_size_in_kb

(Default: 2) A threshold for the total size of all index entries for a partition that Cassandra stores in the partition key cache. If the total size of all index entries for a partition exceeds this amount, Cassandra stops putting entries for this partition into the partition key cache. This limitation prevents index entries from large partitions from taking up all the space in the partition key cache (which is controlled by `key_cache_size_in_mb`).

row_cache_class_name

(Default: disabled - row cache is not enabled)note The classname of the row cache provider to use. Valid values: `OHCProvider` (fully off-heap) or `SerializingCacheProvider` (partially off-heap).

row_cache_keys_to_save

(Default: disabled - all keys are saved)note Number of keys from the row cache to save.

row_cache_size_in_mb

(Default: 0- disabled) Maximum size of the row cache in memory. The row cache can save more time than key_cache_size_in_mb,, but it is space-intensive because it contains the entire row. Use the row cache only for hot rows or static rows. If you reduce the size, you may not get you hottest keys loaded on start up.

row_cache_save_period

(Default: 0- disabled) The number of seconds that rows are kept in cache. Caches are saved to saved_caches_directory. This setting has limited use as described in row_cache_size_in_mb.

Counter caches properties

Counter cache helps to reduce counter locks' contention for hot counter cells. In case of RF = 1 a counter cache hit causes Cassandra to skip the read before write entirely. With RF > 1 a counter cache hit still helps to reduce the duration of the lock hold, helping with hot counter cell updates, but does not allow skipping the read entirely. Only the local (clock, count) tuple of a counter cell is kept in memory, not the whole counter, so it is relatively cheap.

Note: If you reduce the counter cache size, Cassandra may load the hottest keys start-up.

counter_cache_size_in_mb

(Default value: empty)note When no value is set, Cassandra uses the smaller of minimum of 2.5% of Heap or 50MB. If your system performs counter deletes and relies on low gc_grace_seconds, you should disable the counter cache. To disable, set to 0.

counter_cache_save_period

(Default: 7200 seconds [2 hours]) the amount of time after which Cassandra saves the counter cache (keys only). Cassandra saves caches to saved_caches_directory.

counter_cache_keys_to_save

(Default value: disabled)note Number of keys from the counter cache to save. When this property is disabled, Cassandra saves all keys.

Tombstone settings

When executing a scan, within or across a partition, Cassandra must keep tombstones in memory to allow them to return to the coordinator. The coordinator uses tombstones to ensure that other replicas know about the deleted rows. Workloads that generate numerous tombstones may cause performance problems and exhaust the server heap. See Cassandra anti-patterns: Queues and queue-like datasets. Adjust these thresholds only if you understand the impact and want to scan more tombstones. You can adjust these thresholds at runtime using the `StorageServiceMBean`.

Related information: Cassandra anti-patterns: Queues and queue-like datasets

tombstone_warn_threshold

(Default: 1000) Cassandra issues a warning if a query scans more than this number of tombstones.

tombstone_failure_threshold

(Default: 100000) Cassandra aborts a query if it scans more than this number of tombstones.

Network timeout settings

range_request_timeout_in_ms

(Default: 10000 milliseconds) The number of milliseconds that the coordinator waits for sequential or index scans to complete before timing it out.

read_request_timeout_in_ms

(Default: 5000 milliseconds) The number of milliseconds that the coordinator waits for read operations to complete before timing it out.

counter_write_request_timeout_in_ms

(Default: 5000 milliseconds) The number of milliseconds that the coordinator waits for counter writes to complete before timing it out.

cas_contention_timeout_in_ms

(Default: 1000 milliseconds) The number of milliseconds during which the coordinator continues to retry a CAS (compare and set) operation that contends with other proposals for the same row. If the coordinator cannot complete the operation within this timespan, it aborts the operation.

truncate_request_timeout_in_ms

(Default: 60000 milliseconds) The number of milliseconds that the coordinator waits for a truncate (the removal of all data from a table) to complete before timing it out. The long default value allows Cassandra to take a snapshot before removing the data. If auto_snapshot is disabled (not recommended), you can reduce this time.

write_request_timeout_in_ms

(Default: 2000 milliseconds) The number of milliseconds that the coordinator waits for a write operations to complete before timing it out.

Related information: Hinted Handoff: repair during write path on page 158

request_timeout_in_ms

(Default: 10000 milliseconds) The default timeout value for other miscellaneous operations.

Related information: Hinted Handoff: repair during write path on page 158

Inter-node settings

cross_node_timeout

(Default: false) Enables or disables operation timeout information exchange between nodes (to accurately measure request timeouts). If this property is disabled, Cassandra assumes the requests are forwarded to the replica instantly by the coordinator, which means that under overload conditions extra time is required for processing already-timed-out requests.

CAUTION: Before enabling this property make sure NTP (network time protocol) is installed and the times are synchronized among the nodes.

internode_send_buff_size_in_bytes

(Default: N/A)[note] The sending socket buffer size in bytes for inter-node calls.

The buffer size set by this parameter and internode_recv_buff_size_in_bytes is limited by `net.core.wmem_max`. If this property is not set, `net.ipv4.tcp_wmem` determines the buffer size. See man tcp and:

- `/proc/sys/net/core/wmem_max`
- `/proc/sys/net/core/rmem_max`
- `/proc/sys/net/ipv4/tcp_wmem`
- `/proc/sys/net/ipv4/tcp_wmem`

Related information: TCP settings

internode_recv_buff_size_in_bytes

(Default: N/A)[note]The receiving socket buffer size in bytes for inter-node calls.

internode_compression

(Default: all) Controls whether traffic between nodes is compressed. Valid values:

- all

  Compresses all traffic.

- dc

  Compresses traffic between datacenters only.
- none

  No compression.

inter_dc_tcp_nodelay

(Default: false) Enable this property or disable tcp_nodelay for inter-datacenter communication. If this property is disabled, the network sends larger, but fewer, network packets. This reduces overhead from the TCP protocol itself. However, disabling `inter_dc_tcp_nodelay` may increase latency by blocking cross data-center responses.

streaming_socket_timeout_in_ms

(Default: 3600000 - 1 hour)[note] Enables or disables socket timeout for streaming operations. If a streaming times out by exceeding this number of milliseconds, Cassandra retries it from the start of the current file. Setting this value too low can result in a significant amount of data re-streaming.

Native transport (CQL Binary Protocol)

start_native_transport

(Default: true) Enables or disables the native transport server. This server uses the same address as the rpc_address, but the port it uses is different from rpc_port. See native_transport_port.

native_transport_port

(Default: 9042) The port where the CQL native transport listens for clients.

native_transport_max_threads

(Default: 128)[note] The maximum number of thread handling requests. Similar to rpc_max_threads, but this property differs as follows:

- The default for `native_transport_max_threads` is 128; the default for `rpc_max_threads` is unlimited.
- There is no corresponding `native_transport_min_threads`.
- Cassandra stops idle native transport threads after 30 seconds.

native_transport_max_frame_size_in_mb

(Default: 256MB) The maximum allowed size of a frame. Frame (requests) larger than this are rejected as invalid.

native_transport_max_concurrent_connections

(Default: -1) The maximum number of concurrent client connections. The default value of -1 means unlimited.

native_transport_max_concurrent_connections_per_ip

(Default: -1) The maximum number of concurrent client connections per source IP address. The default value of -1 means unlimited.

RPC (remote procedure call) settings

Settings for configuring and tuning client connections.

broadcast_rpc_address

(Default: unset)[note] The RPC address for broadcast to drivers and other Cassandra nodes. This cannot be set to 0.0.0.0. If left blank, Cassandra uses the rpc_address or rpc_interface. If rpc_address or rpc_interfaceis set to 0.0.0.0, this property must be set.

rpc_port

(Default: 9160) Thrift port for client connections.

start_rpc

(Default: true) Enables or disables the Thrift RPC server.

rpc_keepalive

(Default: true) Enables or disables keepalive on client connections (RPC or native).

rpc_max_threads

(Default: unlimited)note Regardless of your choice of RPC server (rpc_server_type), rpc_max_threads dictates the maximum number of concurrent requests in the RPC thread pool. If you are using the parameter sync (see rpc_server_type) it also dictates the number of clients that can be connected. A high number of client connections could cause excessive memory usage for the thread stack. Connection pooling on the client side is highly recommended. Setting a rpc_max_threads acts as a safeguard against misbehaving clients. If the number of threads reaches the maximum, Cassandra blocks additional connections until a client disconnects.

rpc_min_threads

(Default: 16)noteThe minimum thread pool size for remote procedure calls.

rpc_recv_buff_size_in_bytes

(Default: N/A)note The receiving socket buffer size for remote procedure calls.

rpc_send_buff_size_in_bytes

(Default: N/A)note The sending socket buffer size in bytes for remote procedure calls.

rpc_server_type

(Default: sync) Cassandra provides three options for the RPC server. On Windows, `sync` is about 30% slower than `hsha`. On Linux, `sync` and `hsha` performance is about the same, but `hsha` uses less memory.

- sync: (Default: one thread per Thrift connection.)

  For a very large number of clients, memory is the limiting factor. On a 64-bit JVM, 180KB is the minimum stack size per thread and corresponds to your use of virtual memory. Physical memory may be limited depending on use of stack space.
- hsha:

  Half synchronous, half asynchronous. All Thrift clients are handled asynchronously using a small number of threads that does not vary with the number of clients. This mechanism scales well to many clients. The RPC requests are synchronous (one thread per active request).

  Note: If you select this option, you must change the default value (unlimited) of rpc_max_threads.
- Your own RPC server

  You must provide a fully-qualified class name of an `o.a.c.t.TServerFactory` that can create a server instance.

Advanced fault detection settings

Settings to handle poorly performing or failing components.

gc_warn_threshold_in_ms

(Default: 1000) Any GC pause longer than this interval is logged at the WARN level. (By default, Cassandra logs any GC pause greater than 200 ms at the INFO level.)

Additional information: Configuring logging on page 127.

dynamic_snitch_badness_threshold

(Default: 0.1) The performance threshold for dynamically routing client requests away from a poorly performing node. Specifically, it controls how much worse a poorly performing node has to be before the dynamic snitch prefers other replicas over it. A value of 0.2 means Cassandra continues to prefer the static snitch values until the node response time is 20% worse than the best performing node. Until the threshold is reached, incoming requests are statically routed to the closest replica (as determined by the snitch). A value greater than zero for this parameter, with a value of less than 1.0 for read_repair_chance, maximizes cache capacity across the nodes.

dynamic_snitch_reset_interval_in_ms

(Default: 600000 milliseconds) Time interval after which Cassandra resets all node scores. This allows a bad node to recover.

dynamic_snitch_update_interval_in_ms

(Default: 100 milliseconds) The number of milliseconds between Cassandra's calculation of node scores. Because score calculation is CPU intensive, be careful when reducing this interval.

hints_flush_period_in_ms

(Default: 10000) The number of milliseconds Cassandra waits before flushing hints from internal buffers to disk.

hints_directory

(Default: $CASSANDRA_HOME/data/hints) The directory in which hints are stored.

hinted_handoff_enabled

(Default: true) Enables or disables hinted handoff. To enable per datacenter, add a list of datacenters. For example: `hinted_handoff_enabled: DC1,DC2`. A hint indicates that the write needs to be replayed to an unavailable node. Cassandra writes the hint to a hints file on the coordinator node.

Related information: Hinted Handoff: repair during write path on page 158

hinted_handoff_disabled_datacenters

(Default: none) A blacklist of datacenters that will not perform hinted handoffs. To disable hinted handoff on a certain datacenter, , add its name to this list. For example: `hinted_handoff_disabled_datacenters: - DC1 - DC2`.

Related information: Hinted Handoff: repair during write path on page 158

hinted_handoff_throttle_in_kb

(Default: 1024) Maximum amount of traffic per delivery thread in kilobytes per second. This rate reduces proportionally to the number of nodes in the cluster. For example, if there are two nodes in the cluster, each delivery thread uses the maximum rate. If there are three, each node throttles to half of the maximum, since the two nodes are expected to deliver hints simultaneously.

Note: When applying this limit, Cassandra computes the hint transmission rate based on the uncompressed hint size, even if internode_compression or hints_compression is enabled.

max_hint_window_in_ms

(Default: 10800000 milliseconds [3 hours]) Maximum amount of time during which Cassandra generates hints for an unresponsive node. After this interval, Cassandra does not generate any new hints for the node until it is back up and responsive. If the node goes down again, Cassandra starts a new interval. This setting can prevent a sudden demand for resources when a node is brought back online and the rest of the cluster attempts to replay a large volume of hinted writes.

Related information: Failure detection and recovery on page 14

max_hints_delivery_threads

(Default: 2) Number of threads Cassandra uses to deliver hints. In multiple data-center deployments, consider increasing this number because cross data-center handoff is generally slower.

max_hints_file_size_in_mb

(Default: 128) The maximum size for a single hints file, in megabytes.

hints_compression

(Default: LZ4Compressor) The compressor for hint files. Supported compressors: LZ, Snappy, and Deflate. If you do not specify a compressor, Cassandra does not compress hints files.

batchlog_replay_throttle_in_kb

(Default: 1024KB per second) Total maximum throttle for replaying hints. Throttling is reduced proportionally to the number of nodes in the cluster.

Request scheduler properties

Settings to handle incoming client requests according to a defined policy. If your nodes are overloaded and dropping requests, DataStax recommends that you add more nodes rather than use these properties to prioritize requests.

Note: The properties in this section apply only to the Thrift transport. They have no effect on the use of CQL over the native protocol.

request_scheduler

(Default: `org.apache.cassandra.scheduler.NoScheduler`) The scheduler to handle incoming client requests according to a defined policy. This scheduler is useful for throttling client requests in single clusters containing multiple keyspaces. This parameter is specifically for requests from the client and does not affect inter-node communication. Valid values:

- `org.apache.cassandra.scheduler.NoScheduler`

  Cassandra does no scheduling.
- `org.apache.cassandra.scheduler.RoundRobinScheduler`

  Cassandra uses a round robin of client requests to a node with a separate queue for each request_scheduler_id property.
- Cassandra uses a Java class that implements the `RequestScheduler` interface.

request_scheduler_id

(Default: keyspace)[note] The scope of the scheduler's activity. Currently the only valid value is keyspace. See weights.

request_scheduler_options

(Default: disabled) A list of properties that define configuration options for request_scheduler:

- throttle_limit: The number of in-flight requests per client. Requests beyond this limit are queued up until running requests complete. Recommended value is ((concurrent_reads + concurrent_writes) × 2).
- default_weight: (Default: 1)[note]

  How many requests the scheduler handles during each turn of the `RoundRobin`.
- weights: (Default: Keyspace: 1)

  A list of keyspaces. How many requests the scheduler handles during each turn of the `RoundRobin`, based on the request_scheduler_id.

Thrift interface properties

Legacy API for older clients. CQL is a simpler and better API for Cassandra.

thrift_framed_transport_size_in_mb

(Default: 15) Frame size (maximum field length) for Thrift. The frame is the row or part of the row that the application is inserting.

thrift_max_message_length_in_mb

(Default: 16) The maximum length of a Thrift message in megabytes, including all fields and internal Thrift overhead (1 byte of overhead for each frame). Calculate message length in conjunction with batches. A frame length greater than or equal to 24 accommodates a batch with four inserts, each of which is 24 bytes. The required message length is greater than or equal to 24+24+24+24+4 (number of frames).

Security properties

Server and client security settings.

authenticator

(Default: `AllowAllAuthenticator`) The authentication backend. It implements `IAuthenticator` for identifying users. Available authenticators:

- `AllowAllAuthenticator`:

  Disables authentication; Cassandra performs no checks.
- `PasswordAuthenticator`

Authenticates users with user names and hashed passwords stored in the `system_auth.credentials` table. Leaving the default replication factor of 1 set for the *system_auth* keyspace results in denial of access to the cluster if the single replica of the keyspace goes down. For multiple datacenters, be sure to set the replication class to `NetworkTopologyStrategy`.

Related information: About Internal authentication on page 94

internode_authenticator

(Default: enabled)note Internode authentication backend. It implements `org.apache.cassandra.auth.AllowAllInternodeAuthenticator` to allows or disallow connections from peer nodes.

authorizer

(Default: `AllowAllAuthorizer`) The authorization backend. It implements `IAuthenticator` to limit access and provide permissions. Available authorizers:

- `AllowAllAuthorizer`

  Disables authorization: Cassandra allows any action to any user.
- `CassandraAuthorizer`

  Stores permissions in system_auth.permissions table. Leaving the default replication factor of 1 set for the *system_auth* keyspace results in denial of access to the cluster if the single replica of the keyspace goes down. For multiple datacenters, be sure to set the replication class to `NetworkTopologyStrategy`.

Related information: Object permissions on page 99

role_manager

(Default: CassandraRoleManager) Part of the Authentication & Authorization backend that implements `IRoleManager` to maintain grants and memberships between roles. Out of the box, Cassandra provides `org.apache.cassandra.auth.CassandraRoleManager`, which stores role information in the system_auth keyspace. Most functions of the `IRoleManager` require an authenticated login, so unless the configured `IAuthenticator` actually implements authentication, most of this functionality will be unavailable. `CassandraRoleManager` stores role data in the `system_auth` keyspace. If you use the role manager, increase `system_auth` keyspace replication factor .

roles_validity_in_ms

(Default: 2000) Fetching permissions can be an expensive operation depending on the authorizer, so this setting allows flexibility. Validity period for roles cache; set to 0 to disable. Granted roles are cached for authenticated sessions in `AuthenticatedUser` and after the period specified here, become eligible for (async) reload. Disabled automatically for `AllowAllAuthenticator`.

roles_update_interval_in_ms

(Default: 2000) Enable to refresh interval for roles cache. Defaults to the same value as `roles_validity_in_ms`. After this interval, cache entries become eligible for refresh. Upon next access, Cassandra schedules an async reload, and returns the old value until the reload completes. If `roles_validity_in_ms` is non-zero, then this must be also.

credentials_validity_in_ms

(Default: 2000) How many milliseconds credentials in the cache remain valid. This cache is tightly coupled to the provided PasswordAuthenticator implementation of IAuthenticator. If another `IAuthenticator` implementation is configured, Cassandra does not use this cache, and these settings have no effect. Set to 0 to disable.

Related information: Internal authentication on page 94, Internal authorization on page 99

Note: Credentials are cached in encrypted form. This may cause a performance penalty that offsets the reduction in latency gained by caching.

credentials_update_interval_in_ms

(Default: same value as credentials_validity_in_ms) After this interval, cache entries become eligible for refresh. The next time the cache is accessed, the system schedules an asynchronous reload of the cache. Until this cache reload is complete, the cache returns the old values.

If credentials_validity_in_ms is nonzero, this property must also be nonzero.

permissions_validity_in_ms

(Default: 2000) How many milliseconds permissions in cache remain valid. Depending on the authorizer, such as `CassandraAuthorizer`, fetching permissions can be resource intensive. This setting is disabled when set to 0 or when `AllowAllAuthorizer` is set.

Related information: Object permissions on page 99

permissions_update_interval_in_ms

(Default: same value as permissions_validity_in_ms)note If enabled, sets refresh interval for the permissions cache. After this interval, cache entries become eligible for refresh. On next access, Cassandra schedules an async reload and returns the old value until the reload completes. If permissions_validity_in_ms is nonzero, roles_update_interval_in_ms must also be non-zero.

server_encryption_options

Enables or disables inter-node encryption. If you enable server_encryption_options, you must also generate keys and provide the appropriate key and truststore locations and passwords. There are no custom encryption options currently enabled for Cassandra. Available options:

- internode_encryption: (Default: none) Enables or disables encryption of inter-node communication using the TLS_RSA_WITH_AES_128_CBC_SHA cipher suite for authentication, key exchange, and encryption of data transfers. Use the DHE/ECDHE ciphers if running in (Federal Information Processing Standard) FIPS 140 compliant mode. Available inter-node options:
  - all

    Encrypt all inter-node communications.
  - none

    No encryption.
  - dc

    Encrypt the traffic between the datacenters (server only).
  - rack

    Encrypt the traffic between the racks (server only).
- keystore: (Default: `conf/.keystore`)

  The location of a Java keystore (JKS) suitable for use with Java Secure Socket Extension (JSSE), which is the Java version of the Secure Sockets Layer (SSL), and Transport Layer Security (TLS) protocols. The keystore contains the private key used to encrypt outgoing messages.
- keystore_password: (Default: *cassandra*)

  Password for the keystore.
- truststore: (Default: `conf/.truststore`)

  Location of the truststore containing the trusted certificate for authenticating remote servers.
- truststore_password: (Default: *cassandra*)

  Password for the truststore.

The passwords used in these options must match the passwords used when generating the keystore and truststore. For instructions on generating these files, see Creating a Keystore to Use with JSSE.

The advanced settings:

- protocol: (Default: TLS)
- algorithm: (Default: SunX509)

- store_type: (Default: JKS)
- cipher_suites: (Default: TLS_RSA_WITH_AES_128_CBC_SHA,TLS_RSA_WITH_AES_256_CBC_SHA)
- require_client_auth: (Default: false)

  Enables or disables certificate authentication.

Related information: Node-to-node encryption on page 115

client_encryption_options

Enables or disables client-to-node encryption. You must also generate keys and provide the appropriate key and truststore locations and passwords. There are no custom encryption options are currently enabled for Cassandra. Available options:

- enabled: (Default: false)

  To enable, set to true.
- keystore: (Default: `conf/.keystore`)

  The location of a Java keystore (JKS) suitable for use with Java Secure Socket Extension (JSSE), which is the Java version of the Secure Sockets Layer (SSL), and Transport Layer Security (TLS) protocols. The keystore contains the private key used to encrypt outgoing messages.
- keystore_password: (Default: *cassandra*)

  Password for the keystore. This must match the password used when generating the keystore and truststore.
- require_client_auth: (Default: false)

  Enables or disables certificate authentication. (Available starting with Cassandra 1.2.3.)
- truststore: (Default: `conf/.truststore`)

  Set this property if require_client_auth is `true`.
- truststore_password: *truststore_password*

  Set if require_client_auth is `true`.

Advanced settings:

- protocol: (Default: TLS)
- algorithm: (Default: SunX509)
- store_type: (Default: JKS)
- cipher_suites: (Default: TLS_RSA_WITH_AES_128_CBC_SHA,TLS_RSA_WITH_AES_256_CBC_SHA)

Related information: Client-to-node encryption on page 116

transparent_data_encryption_options

Enables encryption of data *at rest* (on-disk). Recommendation: download and install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files for your version of the JDK.

Cassandra supports transparent data encryption for the following file types:

- commitlog
- hints

Available options:

- enabled: (Default: false)
- chunk_length_kb: (Default: 64)
- cipher: options:
  - AES
  - CBC

    - PKCS5Padding
- key_alias: testing:1
- iv_length: 16

  Note: iv_length is commented out in the default `cassandra.yaml` file. Uncomment only if cipher is set to AES. The value must be 16 (bytes).
- key_provider:
    - class_name: `org.apache.cassandra.security.JKSKeyProvider`

      parameters:
        - keystore: conf/.keystore
        - keystore_password: cassandra
        - store_type: JCEKS
        - key_password: cassandra

ssl_storage_port

(Default: 7001) The SSL port for encrypted communication. Unused unless enabled in encryption_options.

native_transport_port_ssl

(Default: 9142) In Cassandra 3.0 and later, an additional dedicated port can be designated for encryption. If client encryption is enabled and `native_transport_port_ssl` is disabled, the `native_transport_port` (default: 9042) will encrypt all traffic. To use both unencrypted and encrypted traffic, enable `native_transport_port_ssl`

Cassandra include file

To set environment variables, Cassandra can use the `cassandra.in.sh` file, located in:

- Tarball installations: *`install_location`*`/bin/cassandra.in.sh`
- Package installations: `/usr/share/cassandra/cassandra.in.sh`

Security

Securing Cassandra

Cassandra provides these security features to the open source community.

- Authentication based on internally controlled rolename/passwords

  Cassandra authentication is roles-based and stored internally in Cassandra system tables. Administrators can create, alter, drop, or list roles using CQL commands, with an associated password. Roles can be created with superuser, non-superuser, and login privileges. The internal authentication is used to access Cassandra keyspaces and tables, and by cqlsh and DevCenter to authenticate connections to Cassandra clusters and sstableloader to load SSTables.
- Authorization based on object permission management

  Authorization grants access privileges to Cassandra cluster operations based on role authentication. Authorization can grant permission to access the entire database or restrict a role to individual table access. Roles can grant authorization to authorize other roles. Roles can be granted to roles. CQL commands GRANT and REVOKE are used to manage authorization.
- Authentication and authorization based on JMX username/passwords

JMX (Java Management Extensions) technology provides a simple and standard way of managing and monitoring resources related to an instance of a Java Virtual Machine (JVM). This is achieved by instrumenting resources with Java objects known as Managed Beans (MBeans) that are registered with an MBean server. JMX authentication stores username and associated passwords in two files, one for passwords and one for access. JMX authentication is used by nodetool and external monitoring tools such as jconsole.In Cassandra 3.6 and later, JMX authentication and authorization can be accomplished using Cassandra's internal authentication and authorization capabilities.

- SSL encryption

  Cassandra provides secure communication between a client and a database cluster, and between nodes in a cluster. Enabling SSL encryption ensures that data in flight is not compromised and is transferred securely. Client-to-node and node-to-node encryption are independently configured. Cassandra tools (cqlsh, nodetool, DevCenter) can be configured to use SSL encryption. The DataStax drivers can be configured to secure traffic between the driver and Cassandra.
- General security measures

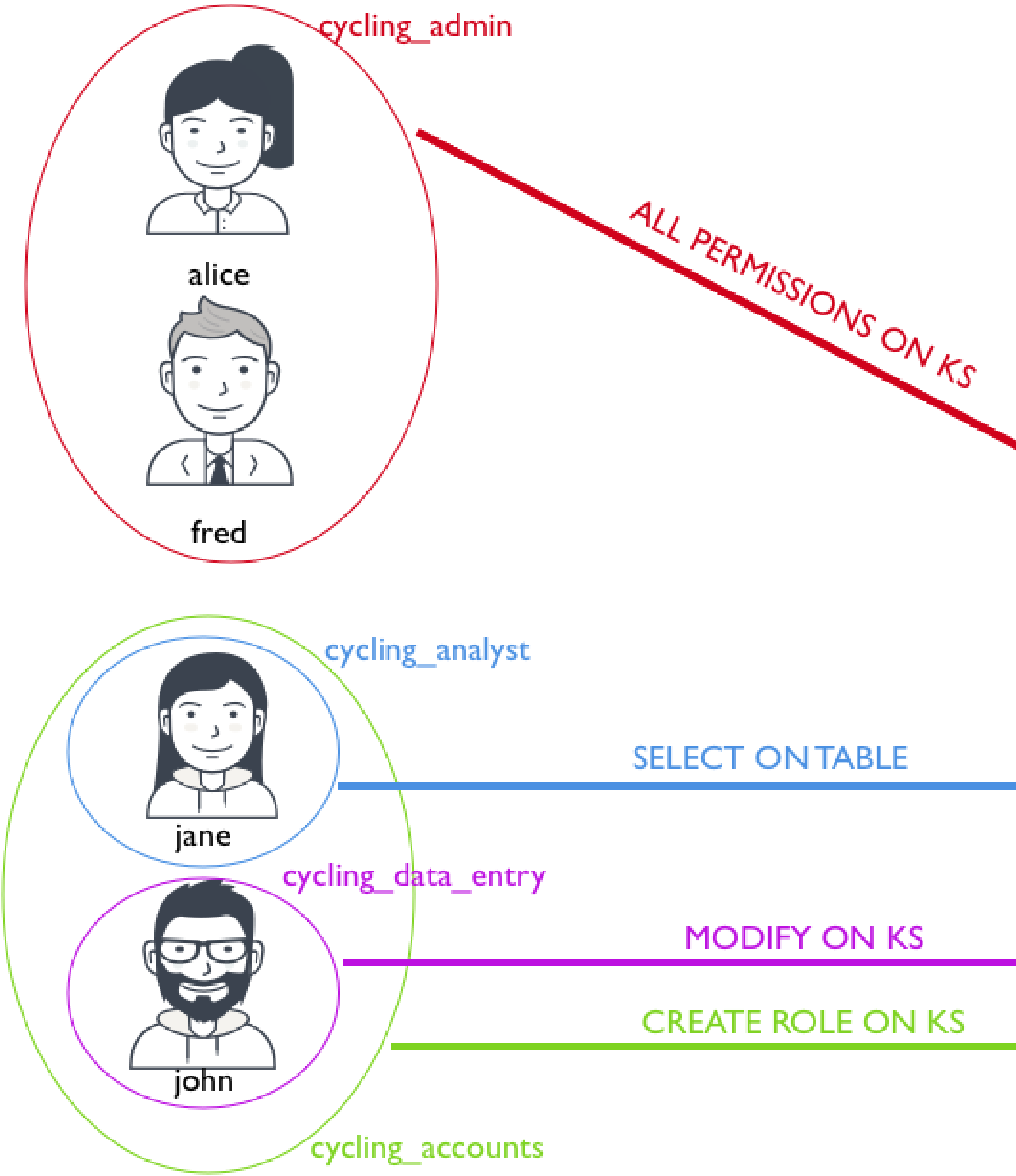
  Typically, production Cassandra clusters will have all non-essential firewall ports closed. Some ports must be open in order for nodes to communicate in the cluster. These ports are detailed.

Internal authentication

About Internal authentication

Like many databases, Cassandra uses rolenames and passwords for internal authentication. Role-based authentication encompasses both users and roles to bring a number of useful features to authorization. Roles can represent either actual individual users or roles that those users have in administering and accessing the Cassandra cluster.

Figure: Roles assigned to individuals and functions



For example, a user named *alice* is created and given login privileges:

```
CREATE ROLE alice WITH PASSWORD = 'enjoyLife' AND LOGIN = true;
```

Note that the user is created as a role and this user can log into the database with the assigned password credentials. Roles can be created with superuser, non-superuser, and login privileges. Superuser privileges allow a role to perform any database operations. Next, we create a role that will have be given access to all the functionality of a particular keyspace:

```
CREATE ROLE cycling_admin WITH PASSWORD = '1234abcd';
GRANT ALL PERMISSIONS ON KEYSPACE cycling TO cycling_admin;
```

This role, when assigned to a user, will provide certain privileges to the user based on the role's privileges; the role that is granted this role will inherit the *cycling_admin* privileges. The *cycling_admin* role is granted all permissions on the keyspace *cycling* in the second command. When *alice* is granted the role *cycling_admin*, *alice* is now granted all permissions on the keyspace *cycling*:

```
GRANT cycling_admin TO alice;
```

An individual user can be granted any number of roles, just as any functional role can be granted another role's permissions. In this example, the role *cycling_analyst* has the ability to select data, and then gains the ability to select data in the another table *hockey* when the role *hockey_analyst* is granted.

```
CREATE ROLE cycling_analyst WITH PASSWORD = 'zyxw9876';
GRANT SELECT ON TABLE cycling.analysis TO cycling_analyst;
CREATE ROLE hockey_analyst WITH PASSWORD = 'Iget2seeAll';
GRANT SELECT ON TABLE hockey.analysis TO hockey_analyst;
GRANT hockey_analyst TO cycling_analyst;
GRANT cyclist_analyst TO jane;
```

If a user then is granted the role of *cycling_analyst* role, that user will be able to select data in the additional table *hockey* The illustration above would be modified to show that the user *jane* now has access to two tables.

Note: Permissions and `SUPERUSER` status are inherited, but the `LOGIN` privilege is not.

An important change that roles-based access control also introduces is that the need for `SUPERUSER` privileges in order to perform user/role management operations is removed. A role can be authorized to create roles or be authorized to grant and revoke permissions:

```
// Give cycling_accounts the right to create roles
GRANT CREATE ON ALL ROLES TO cycling_accounts;
// Give cycling_accounts the right to grant or revoke permissions
GRANT AUTHORIZE ON KEYSPACE cycling TO cycling_accounts;
GRANT cyclist_accounts TO jane;
GRANT cyclist_accounts TO john;
```

Internal authentication and authorization information is stored in the following Cassandra tables:

system_auth.roles

Table that stores the role name, whether the role can be used for login, whether the role is a superuser, what other roles the role may be a member of, and a bcrypt salted hash password for the role.

system_auth.role_members

Table that stores the roles and role members.

system_auth.role_permissions

Table that stores the role, a resource (keyspace, table), and the permission that the role has to access the resource.

system_auth.resource_role_permissons_index

Table that stores the role and a resource that the role has a set permission.

Cassandra is configured with a default superuser role and password pair of *cassandra*/*cassandra* by default. Using this role, additional roles can be created using CQL commands. To secure the system, this default role should be deleted once a non-default superuser has been created.

Once roles and passwords have been set, Cassandra can be configured to use authentication in the cassandra.yaml file.

If roles exist and Cassandra is configured to use authentication, Cassandra tools must be executed with optional authentication options.

- cqlsh with authentication
- DevCenter authenticated connections
- DataStax drivers - produced and certified by DataStax to work with Cassandra.

Configuring authentication

Steps for configuring authentication.

Procedure

1. Change the authenticator option in the cassandra.yaml file to `PasswordAuthenticator`:

   ```
   authenticator: PasswordAuthenticator
   ```

   By default, the authenticator option is set to `AllowAllAuthenticator`.

2. Restart Cassandra.
3. Start `cqlsh` using the default superuser name and password:

   ```
   $  cqlsh -u cassandra -p cassandra
   ```
4. To ensure that the keyspace is always available, increase the replication factor for the *system_auth* keyspace to 3 to 5 nodes per datacenter (recommended):

   ```
   cqlsh> ALTER KEYSPACE "system_auth"
   WITH REPLICATION = {'class' : 'NetworkTopologyStrategy', 'dc1' : 3,
    'dc2' : 2};
   ```

   The `system_auth` keyspace uses a QUORUM consistency level when checking authentication for the default *cassandra* user. For all other users created, superuser or otherwise, a LOCAL_ONE consistency level is used for authenticating.

   CAUTION: Leaving the default replication factor of 1 set for the *system_auth* keyspace results in denial of access to the cluster if the single replica of the keyspace goes down. For multiple datacenters, be sure to set the replication class to `NetworkTopologyStrategy`.
5. After increasing the replication factor of a keyspace, run `nodetool repair` to make certain the change is propagated:

   ```
   $  nodetool repair system_auth
   ```
6. Restart Cassandra.
7. Start `cqlsh` using the superuser name and password:

   ```
   $  cqlsh -u cassandra -p cassandra
   ```
8. To prevent security breaches, replace the default superuser, *cassandra*, with another superuser with a different name:

   ```
   cqlsh> CREATE ROLE <new_super_user> WITH PASSWORD =
    '<some_secure_password>'
       AND SUPERUSER = true
       AND LOGIN = true;
   ```

The default user `cassandra` reads with a consistency level of QUORUM by default, whereas another superuser reads with a consistency level of LOCAL_ONE.

9. Log in as the newly created superuser:

```
$  cqlsh -u <new_super_user> -p <some_secure_password>
```

10. The *cassandra* superuser cannot be deleted from Cassandra. To neutralize the account, change the password to something long and incomprehensible, and alter the user's status to `NOSUPERUSER`:

```
cqlsh> ALTER ROLE cassandra WITH
 PASSWORD='SomeNonsenseThatNoOneWillThinkOf'
    AND SUPERUSER=false;
```

11. Once you create some new roles, you are ready to authorize those roles to access database objects.
12. Fetching role authentication can be a costly operation. To decrease the burden, adjust the validity period for role caching with the roles_validity_in_ms option in the `cassandra.yaml` file (default 2000 milliseconds):

```
roles_validity_in_ms: 2000
```

To disable, set this option to 0. This setting is automatically disabled when the authenticator is set to `AllowAllAuthenticator`.

13. Configure the refresh interval for role caches by setting the roles_update_interval_in_ms option in the `cassandra.yaml` file (default 2000 ms):

```
roles_update_interval_in_ms: 2000
```

If `roles_validity_in_ms` is non-zero, this setting must be set.

Note: The credentials are cached in their encrypted form.

The following steps apply only to Cassandra 3.4 and later:

14. Fetching credentials authentication can be a costly operation. To decrease the burden, adjust the validity period for credential caching with the credentials_validity_in_ms option in the `cassandra.yaml` file (default 2000 ms):

```
credentials_validity_in_ms: 2000
```

To disable, set this option to 0. This setting is automatically disabled when the authenticator is set to `AllowAllAuthenticator`.

15. To set the refresh interval for credentials caches, use the credentials_update_interval_in_ms option (default 2000 ms):

```
credentials_update_interval_in_ms: 2000
```

If `credentials_validity_in_ms` is non-zero, this setting must be set.

16. To disable configuration of authentication and authorization caches (credentials, roles, and permissions) via JMX, uncomment the following line in the `jvm.options` file:

```
#-Dcassandra.disable_auth_caches_remote_configuration=true
```

After setting this option, cache options can only be set in the `cassandra.yaml` file. To make the new setting take effect, restart Cassandra.

Using cqlsh with authentication

Typically, after configuring authentication, logging into cqlsh requires the -u and -p options to the `cqlsh` command. To set credentials for use when launching `cqlsh`, create or modify the `.cassandra/cqlshrc` file. When present, this file passes default login information to `cqlsh`. The cqlshrc.sample file provides an example.

Procedure

1. Create or modify the `cqlshrc` file that specifies a role name and password.

   ```
    [authentication]
   username = fred
   password = !!bang!!$
   ```

   Note: Additional settings in the `cqlshrc` file are described in Creating and using the cqlshrc file.
2. Save the file in *home*`/.cassandra` directory and name it `cqlshrc`.
3. Set permissions on the file to prevent unauthorized access, as the password is stored in plain text. The file must be readable by the user that starts `cassandra`.

   ```
   $ chmod 440 home/.cassandra/cqlshrc
   ```
4. Check the permissions on *home*`/.cassandra/cqlshrc_history` to ensure that plain text passwords are not compromised.

Internal authorization

Object permissions

Object permissions may be assigned using Cassandra's internal authorization mechanism for the following objects:

- keyspace
- table
- function
- aggregate
- roles
- MBeans (in Cassandra 3.6 and later)

Authenticated roles with passwords stored in Cassandra are authorized selective access. The permissions are stored in Cassandra tables.

Permission is configurable for CQL commands `CREATE`, `ALTER`, `DROP`, `SELECT`, `MODIFY`, and `DESCRIBE`, which are used to interact with the database. The `EXECUTE` command may be used to grant permission to a role for the `SELECT`, `INSERT`, and `UPDATE` commands. In addition, the `AUTHORIZE` command may be used to grant permission for a role to `GRANT`, `REVOKE` or `AUTHORIZE` another role's permissions.

Read access to these system tables is implicitly given to every authenticated user or role because the tables are used by most Cassandra tools:

- system_schema.keyspaces
- system_schema.columns
- system_schema.tables
- system.local
- system.peers

Configuring internal authorization

CassandraAuthorizer is one of many possible IAuthorizer implementations. Its advantage is that it stores permissions in the `system_auth.permissions` table to support all authorization-related CQL statements. To activate it, change the `authorizer` option in cassandra.yaml to use the `CassandraAuthorizer`.

Note: To configure authentication, see Internal authentication.

Procedure

1. In the `cassandra.yaml` file, comment out the default `AllowAllAuthorizer` and add the `CassandraAuthorizer`.

   ```
   authorizer: CassandraAuthorizer
   ```

   You can use any authenticator except AllowAll.
2. Increase the replication factor for the `system_auth` keyspace if not already configured.
3. Fetching role permissions can be a costly operation. Role permissions can be cached to decrease the burden. Adjust the validity period for permission caching by setting the permissions_validity_in_ms option in the `cassandra.yaml` file. The default value is 2000 milliseconds. The caching can be disabled by setting the option to 0. This setting is disabled automatically if the authorizer is set to `AllowAllAuthorizer`

   ```
   permissions_validity_in_ms: 2000
   ```
4. A refresh interval for role caches can also be configured by setting the permissions_update_interval_in_ms option in the `cassandra.yaml` file. The default value is the same value as the `permissions_validity_in_ms` setting. If `permissions_validity_in_ms` is non-zero, this setting must be set.

   ```
   permissions_update_interval_in_ms: 2000
   ```

Results

CQL supports these authorization statements:

- GRANT
- LIST PERMISSIONS
- REVOKE

JMX authentication and authorization

JMX Authentication and Authorization

JMX authentication and authorization allows selective users to access JMX tools and JMX metrics. In Cassandra 3.5 and earlier, JMX is configured with password and access files. In Cassandra 3.6 and later, JMX connections can use the same internal authentication and authorization mechanisms as CQL clients.

If usernames and passwords exist and Cassandra is configured to use authentication and authorization, JMX tools must be executed with authentication and authorization options.

- nodetool with authentication
- jconsole with authentication

Enabling JMX authentication and authorization

By default, JMX security is disabled and accessible only from *localhost* without authentication as shown in the following lines from the cassandra-env.sh file:

```
if [ "$LOCAL_JMX" = "yes" ]; then
  JVM_OPTS="$JVM_OPTS -Dcassandra.jmx.local.port=$JMX_PORT -XX:
+DisableExplicitGC"
```

Configuring JMX authentication and authorization can be accomplished using local password and access files to set the usernames, passwords and access permissions. In Cassandra 3.6 and later, Cassandra's internal authentication and authorization can optionally be configured for JMX security.

These two methods work for remote authentication and authorization; the difference is just the location of the configuration settings in the cassandra-env.sh file. Local configuration is placed within the `if ["$LOCAL_JMX" = "yes']; then` block in the file, whereas remote configuration is placed with the `else` block.

Procedure

AUTHENTICATION AND AUTHORIZATION USING LOCAL FILES

- By default, JMX security is disabled and accessible only from *localhost* as shown in the following lines from the cassandra-env.sh file:

  ```
  if [ "$LOCAL_JMX" = "yes" ]; then
    JVM_OPTS="$JVM_OPTS -Dcassandra.jmx.local.port=$JMX_PORT -XX:
  +DisableExplicitGC"
  ```

- Change `$LOCAL_JMX` to `no`. Add the following lines in the remote block in the cassandra-env.sh file:

  ```
  JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.authenticate=true"
    JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.password.file=/etc/
  cassandra/jmxremote.password"
  JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.access.file=/etc/
  cassandra/jmxremote.access"
  ```

- Create a password file and add the user and password for JMX-compliant utilities, specifying the credentials for your environment. The default location of the password file in the cassandra-env.sh is `/etc/cassandra/jmxremote.password`.

  ```
  cassandra cassandra
  <new_superuser> <new_superuser_password>
  <some_other_user> <some_other_user_password>
  controlRole someOtherHardToRememberPassword
  ```

  Important: The default superuser account is a security hazard! This account is used only for the purposes of illustration.

- The password file must be secured from unauthorized readers. Change the ownership of the `jmxremote.password` file to the user who starts `cassandra` and change permissions to read only:

  ```
  $ chown cassandra:cassandra /etc/cassandra/jmxremote.password
  $ chmod 400 /etc/cassandra/jmxremote.password
  ```

  This example presumes that `cassandra` is run by the default user `cassandra`.

- Create an access file and enter the following information. The default location of the access file in the cassandra-env.sh is `/etc/cassandra/jmxremote.access`.

  ```
  cassandra readwrite
  ```

```
<new_superuser> readwrite
<some_other_user> readonly
controlRole readwrite \
create javax.management.monitor.,javax.management.timer. \
unregister
```

Important: The default superuser account is a security hazard! This account is used only for the purposes of illustration.

The `readonly` permission allows the JMX client to read an MBean's attributes and receive notifications. The `readwrite` permission allows the JMX client to set attributes, invoke operations, and create and remove MBeans, in addition to reading an MBean's attributes and receives notifications.

- The access file must be secured from unauthorized readers. Change the ownership of the `jmxremote.access` file to the user who starts `cassandra` and change permissions to read only:

```
$ chown cassandra:cassandra /etc/cassandra/jmxremote.access
$ chmod 400 /etc/cassandra/jmxremote.access
```

  This example presumes that `cassandra` is run by the default user `cassandra`.
- Restart Cassandra to make the change effective.
- Check that `nodetool status` requires the username and password in order to execute. The command should fail without authentication if everything is configured correctly.

```
$ nodetool status
```

- Run `nodetool status` with the cassandra user and password.

```
$ nodetool -u cassandra -pw cassandra status
```

AUTHENTICATION AND AUTHORIZATION WITH CASSANDRA INTERNALS - CASSANDRA 3.6 AND LATER

- By default, JMX security is disabled and accessible only from *localhost* as shown in the following lines from the cassandra-env.sh file:

```
if [ "$LOCAL_JMX" = "yes" ]; then
  JVM_OPTS="$JVM_OPTS -Dcassandra.jmx.local.port=$JMX_PORT -XX:
+DisableExplicitGC"
```

- Comment out the existing line and add or uncomment the following lines in either the local or remote block in the cassandra-env.sh file:

```
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.authenticate=true"
JVM_OPTS="$JVM_OPTS -Dcassandra.jmx.remote.login.config=CassandraLogin"'
JVM_OPTS="$JVM_OPTS -Djava.security.auth.login.config=$CASSANDRA_HOME/
conf/cassandra-jaas.config"
JVM_OPTS="$JVM_OPTS -
Dcassandra.jmx.authorizer=org.apache.cassandra.auth.jmx.AuthorizationProxy"
```

- And comment out the following lines in the cassandra-env.sh file:

```
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.password.file=/etc/
cassandra/jmxremote.password"
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.access.file=/etc/
cassandra/jmxremote.access"
```

- Change authentication in the `cassandra.yaml` file to `PasswordAuthenticator`.

```
authenticator: PasswordAuthenticator
```

- Change authorization in the `cassandra.yaml` file to `CassandraAuthorizer`.

  ```
  authorizer: CassandraAuthorizer
  ```

- Restart Cassandra to make the change effective.
- Check that `nodetool status` requires the username and password in order to execute. The command should fail without authentication if everything is configured correctly.

  ```
  $ nodetool -u cassandra -pw cassandra status
  ```

- Cassandra authorization can be used to grant and revoke permissions to database objects, including MBeans.

SPECIFYING JMX AUTHENTICATION ON COMMAND LINE

- Generally, JMX settings are inserted into the cassandra-env.sh file. However, these options can be specified at the command line:

  ```
  cassandra -Dcom.sun.management.jmxremote.authenticate=true
    -Dcom.sun.management.jmxremote.password.file=/etc/cassandra/
  jmxremote.password
  ```

Example

If you run nodetool status without user and password when authentication and authorization are configured, you'll see an error similar to:

```
Exception in thread "main" java.lang.SecurityException: Authentication failed!
 Credentials required
at
 com.sun.jmx.remote.security.JMXPluggableAuthenticator.authenticationFailure(Unknown
 Source)
at com.sun.jmx.remote.security.JMXPluggableAuthenticator.authenticate(Unknown
 Source)
at sun.management.jmxremote.ConnectorBootstrap
$AccessFileCheckerAuthenticator.authenticate(Unknown Source)
at javax.management.remote.rmi.RMIServerImpl.doNewClient(Unknown Source)
at javax.management.remote.rmi.RMIServerImpl.newClient(Unknown Source)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
at java.lang.reflect.Method.invoke(Unknown Source)
at sun.rmi.server.UnicastServerRef.dispatch(Unknown Source)
at sun.rmi.transport.Transport$1.run(Unknown Source)
at sun.rmi.transport.Transport$1.run(Unknown Source)
at java.security.AccessController.doPrivileged(Native Method)
at sun.rmi.transport.Transport.serviceCall(Unknown Source)
at sun.rmi.transport.tcp.TCPTransport.handleMessages(Unknown Source)
at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run0(Unknown Source)
at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(Unknown Source)
at java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source)
at java.lang.Thread.run(Unknown Source)
at sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(Unknown
 Source)
at sun.rmi.transport.StreamRemoteCall.executeCall(Unknown Source)
at sun.rmi.server.UnicastRef.invoke(Unknown Source)
at javax.management.remote.rmi.RMIServerImpl_Stub.newClient(Unknown Source)
at javax.management.remote.rmi.RMIConnector.getConnection(Unknown Source)
at javax.management.remote.rmi.RMIConnector.connect(Unknown Source)
at javax.management.remote.JMXConnectorFactory.connect(Unknown Source)
at org.apache.cassandra.tools.NodeProbe.connect(NodeProbe.java:146)
at org.apache.cassandra.tools.NodeProbe.<init>(NodeProbe.java:116)
```

```
at org.apache.cassandra.tools.NodeCmd.main(NodeCmd.java:1099)
```

Using nodetool with authentication

After configuring JMX authentication, using `nodetool` requires the -u and -p options to the `nodetool` commands.

Procedure

Run `nodetool` using a pre-configured JMX username and password for `<username>` and `<password>`:

```
$ nodetool -u <username> -pw <password>
```

For Cassandra 3.6 and later, the username and password can be an internally configured Cassandra role and password.

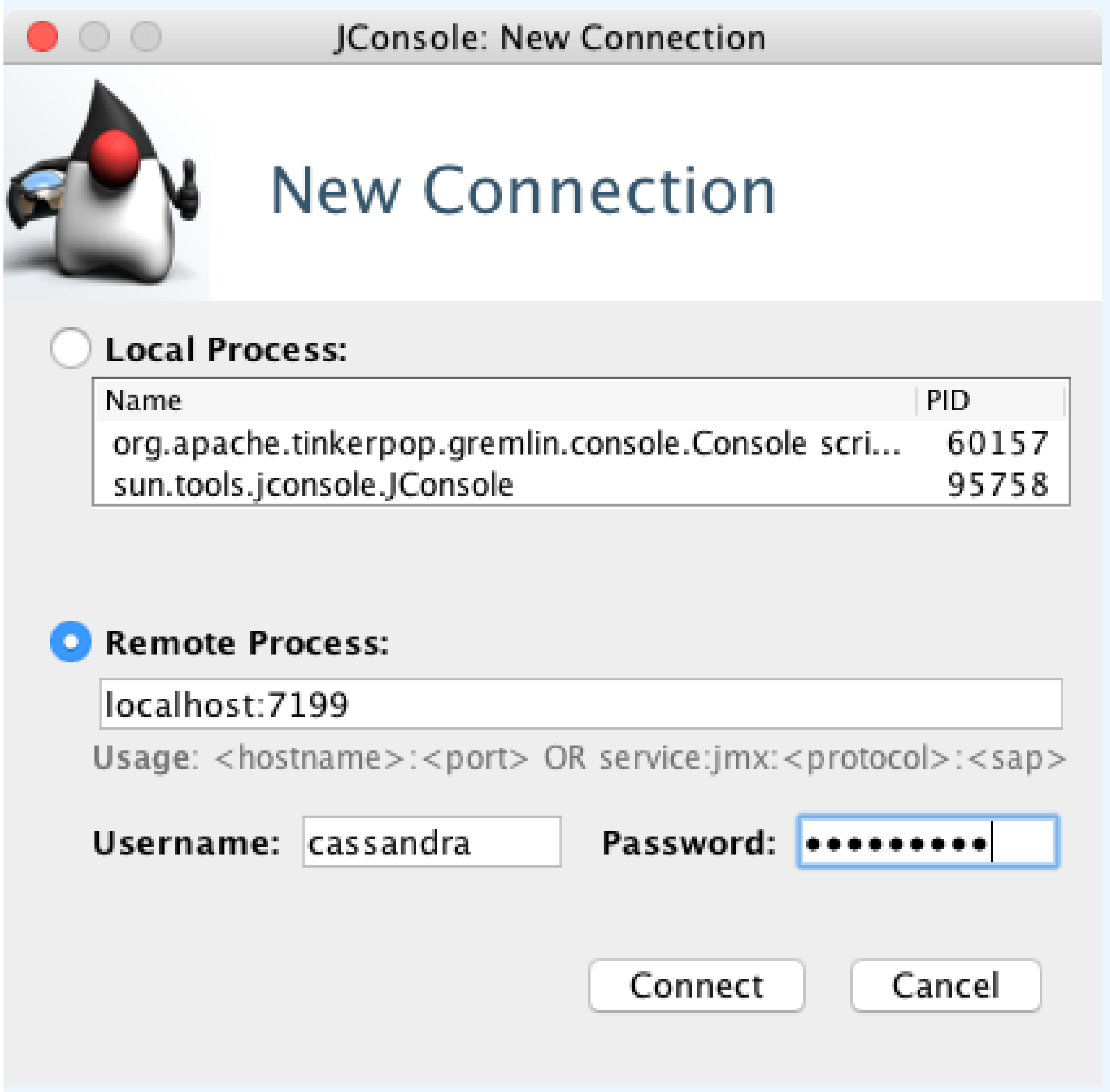
Note: In Cassandra 3.0.8 and later, a user designated `readonly` access can run `nodetool info` so that cluster monitoring is available. In earlier versions, the user must have `readwrite` permission.

Using jconsole with authentication

After configuring JMX authentication, using `jconsole` requires a username and password to complete the remote connection to the Cassandra cluster. Use an appropriate username/password combination.

Procedure

Start `jconsole` using a pre-configured JMX username and password for `<username>` and `<password>`:



JConsole: New Connection

New Connection

○ Local Process:

Name	PID
org.apache.tinkerpop.gremlin.console.Console scri...	60157
sun.tools.jconsole.JConsole	95758

● Remote Process:

localhost:7199

Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap>

Username: cassandra

Password: ●●●●●●●●●

For Cassandra 3.6 and later, the username and password can be an internally configured Cassandra role and password.

SSL encryption

Encrypting Cassandra with SSL

The Secure Socket Layer (SSL) is a cryptographic protocol used to secure communications between computers. For reference, see SSL in wikipedia. Data is encrypted during communication to prevent accidental or deliberate attempts to read the data.

Briefly, SSL works in the following manner. Two entities, either software or hardware, that are communicating with one another. The entities an be a client and node or peers in a cluster. These entities must exchange information to set up trust between them. Each entity that will provide such information must have a generated key that consists of a private key that only the entity stores and a public key that can be exchanged with other entities. If the client wants to connect to the server, the client requests the secure connection and the server sends a certificate that includes its public key. The client checks the validity of the certificate by exchanging information with the server, which the server validates with its private key. If a two-way validation is desired, this process must be carried out in both directions. Private keys and certificates are stored in the *keystore* and public keys are stored in the *truststore*. For systems using a Certificate Authority (CA), the truststore can store certificates signed by the CA for verification. Both keystores and truststores have passwords assigned, referred to as the *keypass* and *storepass*.

Apache Cassandra provides these SSL encryption features for .

- Node-to-node encrypted communication

  Node-to-node, or internode, encryption is used to secure data passed between nodes in a cluster.
- Client-to-node encrypted communication

  Client-to-node encryption is used to secure data passed between a client program, such as cqlsh, DevCenter, or nodetool, and the nodes in the cluster.

Installing Java Cryptography Extension (JCE) Files

Installing the JCE Unlimited Strength Jurisdiction Policy Files can ensure support for all encryption algorithms when using Oracle Java with SSL on Apache Cassandra, and it highly recommended. The files must be installed on every node in the Cassandra cluster.

Some of the cipher suites in the default cassandra.yaml are included only in the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files. To ensure support for all encryption algorithms, install the JCE Unlimited Strength Jurisdiction Policy Files.

Install the JCE files using the appropriate method for your Cassandra installation:

Procedure

Installing the JCE on RHEL-based systems

- Install the EPEL repository:

```
$ sudo yum install epel-release
```

Installing the JCE on Debian-based systems

- Install JCE using webupd8 PPA repository:

```
$ sudo apt-get install oracle-java8-unlimited-jce-policy
```

Installing the JCE using the Oracle jar files

- Download the Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files from Oracle Java SE download page.
- Unzip the downloaded file.
- Copy `local_policy.jar` and `US_export_policy.jar` to the `$JAVA_HOME/jre/lib/security` directory to overwrite the existing jar files.

Preparing server certificates for development

To use SSL encryption for client-to-node encryption or node-to-node encryption, SSL certificates must be generated using keytool. If you generate the certificates for one type of encryption, you do not need to generate them again for the other; the same certificates are used for both. All nodes must have all the relevant SSL certificates on all nodes. A keystore contains private keys. The truststore contains SSL certificates for each node. The certificates in the truststore don't require signing by a trusted and recognized public certification authority.

Procedure

- Generate a private and public key pair on each node of the cluster. Use an alias that identifies the node. Prompts for the keystore password, dname (first and last name, organizational unit, organization, city, state, country), and key password. The dname should be generated with the CN value as the IP address or FQDN for the node.

```
$  keytool -genkey -keyalg RSA -alias node0 -validity 36500 -keystore
 keystore.node0
```

Note: In this example, the value for `--validity` gives this key pair a validity period of 100 years. The default `validity` value for a key pair is 90 days.

- The generation command can also include all prompted-for information in the command line. This example uses an alias of `node0`, a keystore name of `keystore.node0`, uses the same password of `cassandra` for both the keystore and the key, and a dname that identifies the IP address of node0 as `172.31.10.22`.

```
$  keytool -genkey -keyalg RSA -alias node0 -keystore keystore.node0 -
storepass cassandra -keypass cassandra -dname "CN=172.31.10.22, OU=None,
 O=None, L=None, C=None"
```

- Export the public part of the certificate to a separate file.

```
$  keytool -export -alias cassandra -file node0.cer -keystore .keystore
```

- Add the `node0.cer` certificate to the node0 truststore of the node using the `keytool -import` command.

```
$  keytool -import -v -trustcacerts -alias node0 -file node0.cer -keystore
 truststore.node0
```

- `cqlsh` does not work with the certificate in the format generated. `openssl` is used to generate a PEM file of the certificate with no keys, `node0.cer.pem`, and a PEM file of the key with no certificate, `node0.key.pem`. First, the keystore is imported in PKCS12 format to a destination keystore, `node0.p12`, in the example. This is followed by the two commands that extract the two PEM files.

```
$  keytool -importkeystore -srckeystore keystore.node0 -destkeystore
 node0.p12 -deststoretype PKCS12 -srcstorepass cassandra -deststorepass
 cassandra
openssl pkcs12 -in node0.p12 -nokeys -out node0.cer.pem -passin
 pass:cassandra
openssl pkcs12 -in node0.p12 -nodes -nocerts -out node0.key.pem -passin
 pass:cassandra
```

- For client-to-remote-node encryption or node-to-node encryption, use a copying tool such as `scp` to copy the `node0.cer` file to each node. Import the file into the truststore after copying to each node. The example imports the certificate for node0 into the truststore for node1.

```
$  keytool -import -v -trustcacerts -alias node0 -file node0.cer -keystore
 truststore.node1
```

- Make sure keystore file is readable only to the Cassandra daemon and not by any user of the system.
- Check that the certificates exist in the keystore and truststore files using `keytool -list`. The example shows checking for the node1 certificate in the keystore file and for the node0 and node1 certificates in the truststore file.

```
$  keytool -list -keystore keystore.node1
keytool -list -keystore truststore.node1
```

- Import the user's certificate into every node's truststore using keytool:

```
$  keytool -import -v -trustcacerts -alias <username> -file <certificate
 file> -keystore .truststore
```

Preparing SSL certificates for production

To use SSL encryption for client-to-node encryption or node-to-node encryption, SSL certificates must be generated using openssl and keytool. To validate the certificates, a self-signed Certificate Authority (CA) can be generated for production use with Apache Cassandra. The certificates generated using these instructions can be used for both internode and client-to-node encryption. For internode encryption, all nodes must have the truststore that provides the chain of trust for the CA. The certificates in the truststore can either be signed by the self-signed certificate authority used here or by a trusted and recognized public certificate authority.

Procedure

Create a root CA certificate and key

1. Create the root CA certificate and key using `openssl req`. This command uses a certificate configuration file `gen_rootCa_cert.conf`.

```
$ openssl req
    -config gen_rootCa_cert.conf
    -new -x509 -nodes
    -subj /CN=rootCa/OU=TestCluster/O=YourCompany/C=US/
    -keyout rootCa.key
    -out rootCa.crt
    -days 365
```

```
# gen_rootCa_cert.conf
[ req ]
distinguished_name        = req_distinguished_name
prompt        = no
output_password     = myPass
default_bits  = 2048

[ req_distinguished_name ]
C       = US
O       = YourCompany
OU      = TestCluster
CN       = rootCa
```

Option	Description
-config	Configuration file to use
-new	Generate new certificate
-x509	Outputs a self-signed certificate
-nodes	If a private key is created, it is not encrypted
-subj	Sets subject name when processing
-keyout	Specify the private key filename to write
-out	Specify the certificate filename to write
-days	Specify the number of days for which to certify the certificate

The resulting files are the rootCA certificate and the rootCa private key.

An example of the `rootCa.crt` file:

```
-----BEGIN CERTIFICATE-----
MIIDADCCAegCCQCWl1PhaMCqNDANBgkqhkiG9w0BAQsFADBCMQswCQYDVQQGEwJV
UzEMMAoGA1UEChMDTExQMRQwEgYDVQQLEwtUZXN0Q2x1c3RlcjEPMA0GA1UEAxMG
cm9vdENhMB4XDTE2MDkxMDAzNTkzOFoXDTE3MDkxMDAzNTkzOFowQjELMAkGA1UE
BhMCVVMxDDAKBgNVBAoTA0xMUDEUMBIGA1UECxMLVGVzdENsdXN0ZXIxDzANBgNV
BAMTBnJvb3RDYTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBANmFeE58
eafNDPOYr6yFGNHxnwtiKRIs1CXF8pJQ1smgzM2v6D9UeixdZ5bDs0grK14a4UoE
939ZCWKOLcgFi3394XwXHCf0mSjudrHd0ptAQKVSMqILRiJ5nJaR4yqfZBFbB2fl
iQ5r9z2P20zTjlbbXZoJT83KF4Q+ke+VLLmEgSsLowjmq+JPP4Uzlp8dCyVLpAeD
snf/T7/RaeIuZ+Wzje5pMlerY7Cv9A6te/SkFK1bZYzMsTCFOY6RBk1KdcVyDvhi
co1b8Hv5hQLZ4v3nd6RtfmXNdL7YrqnLA9LnmS9ZFIk1w95Plg6hdBOuGT62W3ll
IypMbZBOdWBMp0MCAwEAATANBgkqhkiG9w0BAQsFAAOCAQEAmoh6xkoa71yuVxJO
O24wDfSNIpgAiP1uj7tvgza0yPs221o8p2e/34wdRaWdzLnc3Iu8cLpommuq9b82
/WQNxdqFIJyyJwDTZUZ6VisSSXktDsW3mDPy10As+HJHuQ9adTsi/sOerh85FjmT
BYbTDjX6BsIrwywgFBnb6uud2/GKlzTtlsi5LFLWKHVryxqng+ja4CZZbQ6/GT02
hdP2d/17gGgCi1hIg5KJv/MoVhN0dLb6cueqfxLOOLGkqkXev2NiONzpjQITRwoF
1NUo45DRHi25PAJ8+slUzvii4ADbe3P+SfEI6AETdnadCnA+WOffHI35OqxxePO9
VqWu8w==
-----END CERTIFICATE-----
```

An example of the `rootCa.key` file:

```
-----BEGIN PRIVATE KEY-----
MIIEvgIBADANBgkqhkiG9w0BAQEFAASCBKgwggSkAgEAAoIBAQDZhXhOfHmnzQzz
mK+shRjR8Z8LYikSLNQlxfKSUNbJoMzNr+g/VHosXWeWw7NIKyteGuFKBPd/WQli
ji3IBYt9/eF8Fxwn9Jko7nax3dKbQEClUjKiC0YieZyWkeMqn2QRWwdn5YkOa/c9
j9tM045W212aCU/NyheEPpHvlSy5hIErC6MI5qviTz+FM5afHQslS6QHg7J3/0+/
0WniLmfls43uaTJXq2Owr/QOrXv0pBStW2WMzLEwhTmOkQZNSnXFcg74YnKNW/B7
+YUC2eL953ekbX5lzXS+2K6pywPS55kvWRSJNcPeT5YOoXQTrhk+tlt5ZSMqTG2Q
TnVgTKdDAgMBAAECggEAYsdSn8m08TeTtxdSR3TVlZk00VWNMxy6ZkDi7ADb2Qo1
lv5X6FJzfKwZ+4P5aT95XS21uwhQYDtNoLzSG3AxLVDaUaCo/5f66XSI4DLMjgX6
lVijd6TI/6TcMCAl2dgx+BOvZEX/HFZ5GzK1ssirbdQGSIoL/HbWgQ5s9TB38/Ja
205ZB1lMUVsqdmY2M6vwY0xgr/xKXMgA3KvsW5PFBX1bgh8T6OLuIXTERYezGU4c
c71JJt+1ejHSDInEEIV1peJer6qXopHDMQOcOSTzcZkGgDlFB7DHyGZM1ZsDRDOO
f7Kq02A2c6CwJQeyGyRVIxDO6Ef73zw6UEtWzYCHeQKBgQDw9i8y0nlLgBLZaWor
bJzJN9ML7CfOVw/RuGD2Y0z7su8SWQLLecVW1lwcaec80rD7e/SBr9Q/Kz68va13
sVBx43QEDNp63g36bX917gu/hwBQs6RJlHGJmiNa/p5S01MaMM2YrX35gxBCQJHN
hiE0yBzepfdGDELrEtOcttfCFwKBgQDnGMhpCpBEvYX43l9PQmnJ+P2w3lZoo8RF
YUdpUuyH9n/mbopU3zu5f1roJefiEaxOozuL1sUOCsCN8qK2B3YCcX0LUvLrpAqR
UD+So+eAF9tgFKHDvPLO7YA3iGPUl00cnwl2UXxV+7SONJRZsCbhbGO+T6QWq6YT
eohMfvgbtQKBgFHPlAjSUyJiMoQkeUqTDsx2qq4SmRVCk/lle25MGrgecXMuS3eg
OXMZRp7TChKpijNoS4S4mPx1h1B3qezIhAKW8i3p20f6Go7bHHqCvvRhNqcvxujA
gKfycGyVpFWEsGNlDHj49pt/d0a3O4mnL6EHDF4/xSvAP/wmITjFD44zAoGBALbB
jpwjUnxKNUze7xjLOMYVNutMqaEPAgSsLcFJZu0PL46YFKWR9LV51faJI5xQxada
x5iLPEMilaysGalCtTyxa2YtLxbTH9hTUjMxk75lH4QYTOVy48JpaGCCaBCTptPf
oagEQQPujpd3VWqoN9dF1IuIiAe1rxzwZiG4t5WRAoGBAKdypl5nZgtghlUxoazS
CXfQsIT7g4y0LvoL9+EqdPk98Wl2Cb6MD1M89UqFhoyh65xE73EssGqDyypgYFGE
HS/sMt9PP44ftfWRgQEGje6tJdKXLyUHSF+kKg4mormriOSm54sZD7Qk5RxEVcMq
arKAClJVFkL9ARoAxRQWwidv
-----END PRIVATE KEY-----
```

Verify the rootCa certificate

2. Verify the rootCa certificate.

```
$  openssl x509
    -in rootCa.crt
    -text
    -noout
```

Option	Description
-in	Specify the certificate filename to verify
-text	Print out the certificate in text form including the public key, signature algorithms, issuer and subject names, serial number any extensions present and any trust settings
-noout	Prevents output of the encoded version of the request

This command prints output to the console that is similar to this example:

```
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 10851234054762703412 (0x969753e168c0aa34)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, O=YourCompany, OU=TestCluster, CN=rootCa
        Validity
            Not Before: Sep 10 03:59:38 2016 GMT
            Not After : Sep 10 03:59:38 2017 GMT
        Subject: C=US, O=YourCompany, OU=TestCluster, CN=rootCa
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:d9:85:78:4e:7c:79:a7:cd:0c:f3:98:af:ac:85:
                    18:d1:f1:9f:0b:62:29:12:2c:d4:25:c5:f2:92:50:
                    d6:c9:a0:cc:cd:af:e8:3f:54:7a:2c:5d:67:96:c3:
                    b3:48:2b:2b:5e:1a:e1:4a:04:f7:7f:59:09:62:8e:
                    2d:c8:05:8b:7d:fd:e1:7c:17:1c:27:f4:99:28:ee:
                    76:b1:dd:d2:9b:40:40:a5:52:32:a2:0b:46:22:79:
                    9c:96:91:e3:2a:9f:64:11:5b:07:67:e5:89:0e:6b:
                    f7:3d:8f:db:4c:d3:8e:56:db:5d:9a:09:4f:cd:ca:
                    17:84:3e:91:ef:95:2c:b9:84:81:2b:0b:a3:08:e6:
                    ab:e2:4f:3f:85:33:96:9f:1d:0b:25:4b:a4:07:83:
                    b2:77:ff:4f:bf:d1:69:e2:2e:67:e5:b3:8d:ee:69:
                    32:57:ab:63:b0:af:f4:0e:ad:7b:f4:a4:14:ad:5b:
                    65:8c:cc:b1:30:85:39:8e:91:06:4d:4a:75:c5:72:
                    0e:f8:62:72:8d:5b:f0:7b:f9:85:02:d9:e2:fd:e7:
                    77:a4:6d:7e:65:cd:74:be:d8:ae:a9:cb:03:d2:e7:
                    99:2f:59:14:89:35:c3:de:4f:96:0e:a1:74:13:ae:
                    19:3e:b6:5b:79:65:23:2a:4c:6d:90:4e:75:60:4c:
                    a7:43
                Exponent: 65537 (0x10001)
    Signature Algorithm: sha256WithRSAEncryption
         9a:88:7a:c6:4a:1a:ef:5c:ae:57:12:4e:3b:6e:30:0d:f4:8d:
         22:98:00:88:fd:6e:8f:bb:6f:83:36:b4:c8:fb:36:db:5a:3c:
         a7:67:bf:df:8c:1d:45:a5:9d:cc:b9:dc:dc:8b:bc:70:ba:68:
         9a:6b:aa:f5:bf:36:fd:64:0d:c5:da:85:20:9c:b2:27:00:d3:
         65:46:7a:56:2b:12:49:79:2d:0e:c5:b7:98:33:f2:d7:40:2c:
         f8:72:47:b9:0f:5a:75:3b:22:fe:c3:9e:ae:1f:39:16:39:93:
         05:86:d3:0e:35:fa:06:c2:2b:c3:2c:20:14:19:db:ea:eb:9d:
         db:f1:8a:97:34:ed:96:c8:b9:2c:52:d6:28:75:6b:cb:1a:a7:
         83:e8:da:e0:26:59:6d:0e:bf:19:3d:36:85:d3:f6:77:fd:7b:
         80:68:02:8b:58:48:83:92:89:bf:f3:28:56:13:74:74:b6:fa:
         72:e7:aa:7f:12:ce:38:b1:a4:aa:45:de:bf:63:62:38:dc:e9:
         8d:02:13:47:0a:05:d4:d5:28:e3:90:d1:1e:2d:b9:3c:02:7c:
         fa:c9:54:ce:f8:a2:e0:00:db:7b:73:fe:49:f1:08:e8:01:13:
         76:76:9d:0a:70:3e:58:e7:df:1c:8d:f9:3a:ac:71:78:f3:bd:
          56:a5:ae:f3
```

Generate public/private key pair and keystore for each node

3. Repeat this command for each node. The files can be generated on a single node and distributed out to the nodes after the entire process is completed.

```
keytool -genkeypair
    -keyalg RSA
    -alias 10.200.175.15
    -keystore 10.200.175.15.jks
    -storepass myKeyPass
    -keypass myKeyPass
    -validity 365
    -keysize 2048
    -dname "CN=10.200.175.15, OU=TestCluster, O=YourCompany, C=US"
```

Option	Description
-genkeypair	Command to generate a public/private key pair
-keyalg	Specify the key algorithm
-alias	Assign an unique alias by which keystore entry is accessed
-keystore	Specify the keystore filename
-storepass	Specify the keystore password
-keypass	Specify the private key password
-validity	Specify the number of days for the keystore certificate validity
-keysize	Specify the size of the generated key
-dname	Specify the X.500 Distinguished Name to be associated with the value of alias

In this example, the node IP address is `10.200.175.15` and the keystore filename incorporates that IP address with a suffix of `jks` (Java KeyStore). While the keystore can be named with any convention, the examples here use the IP address in order to map the files to the nodes. The `dname` sets the `CN` value to the node's IP address or FQDN as well. The `storepass` and `keypass` must be the same value.

Check certificates

4. The certificates can be checked once generated:

```
keytool -list
    -keystore 10.200.175.15.jks
    -storepass myKeyPass
```

An example keystore file:

```
Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

10.200.175.15, Sep 10, 2016, PrivateKeyEntry,
Certificate fingerprint (SHA1):
 D6:1B:6C:FE:E3:3B:4B:3D:E0:F0:38:EA:54:AD:F0:E7:1E:D4:CB:4D
```

Export certificate signing request (CSR) for each node

5. Once the node certificate and key are generated, a certificate signing request (CSR) is exported. The CSR will be signed with the rootCa certificate to verify that the node's certificate is trusted.

```
keytool -certreq
    -keystore 10.200.175.15.jks
    -alias 10.200.175.15
    -file 10.200.175.15.csr
    -keypass myKeyPass
    -storepass myKeyPass
    -dname "CN=10.200.175.15, OU=TestCluster, O=YourCompany, C=US"
```

Option	Description
-certreq	Command to export a CSR
-file	Specify the CSR filename
-alias	Assign an unique alias by which keystore entry is accessed
-keystore	Specify the keystore filename
-storepass	Specify the keystore password
-keypass	Specify the private key password
-dname	Specify the X.500 Distinguished Name to be associated with the value of alias

An example CSR file:

```
-----BEGIN NEW CERTIFICATE REQUEST-----
MIICvDCCAaQCAQAwRzELMAkGA1UEBhMCVVMxDDAKBgNVBAoTA0xMUDESMBAGA1UECxMJTExQMDkw
NzE2MRYwFAYDVQQDEw0xMC4yMDAuMTc1LjE1MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKC
AQEAhUJWFQ+9Xgs8KfhLYCBi8i0csY44YkCBFAz3jh6g80wdCqYetYcNxWi
+3rgAFL6BAibt23xG
qBKtNszxyUu8M7ocfw+dVeT4YQJWE0TNAnwlijx
+jyl2IKCFtuON3NmGvvHviPDyNyz2VuB7jlA3
fGuCSsXEzpDXdStKZNcozeIxnmLRv2VKkp8edOX0Bi3QOxwKgwzZQ0/
Z7yWaixl1WB2YKzY9s1Bn
oEoRHdFeQZR7f9QuQYBwIKsCx3cFsiEQJnpKqbZYLFrPmpMrR7ynLx6MGZVDuLfrXp/
yFyuHmamI
M80CGvKljwr/
onalDY2F5AeSdUyBM3vVTWrC9n9jwQIDAQABoDAwLgYJKoZIhvcNAQkOMSEwHzAd
BgNVHQ4EFgQUSbrkacD4XXCIaJQtX+7/
en3Z8r8wDQYJKoZIhvcNAQELBQADggEBAAxU2FbQxy21
EHcnfC4YETDQvXuwv4qQzcT61faZEjmznZ9ekm8ckzV5NlHpyArk12EO13twh94U56ZItKfiYOKW
tP5wUGcoYhdyQnG0p6wunVRJAMidoZMAkjJ6bXwDwgNqnlKL48iGD8ZnguVUM353KwTDm1mvJ4yE
ssEcnBKd4lDzfZN+yx3pmyCr/
MjMCODLF7VVMRH5FpQZ0+uGAIq0fx8FeEugxGie4tkzqP3xkkUB
RDKkfrUC8Z61gL3K1LpLZ77a1okpP3cNkvSStVgbhLH9qwnhCORNGHy+NyZLm1a
+hS4QCAJzRKlC
nsdwUTp+HXUtyNLd7GJHGLPu0YY=
-----END NEW CERTIFICATE REQUEST-----
```

Sign node certificate with rootCa for each node

6. The CSR is input, signed with the rootCa certificate and a signed node certificate is created.

```
openssl x509
    -req
    -CA rootCa.crt
    -CAkey rootCa.key
    -in 10.200.175.15.csr
    -out 10.200.175.15.crt_signed
    -days 365
    -CAcreateserial
    -passin pass:myPass
```

Option	Description
-req	Specify that the input file is a CSR
-CA	Identify the rootCa certificate
-CAkey	Identify the rootCa key
-in	Specify the input filename from which to read a certificate
-out	Specify the output filename for the signed certificate
-CAcreateserial	Specify that a CA serial number file is created if it does not exist
-days	Specify the number of days for which to certify the certificate
-passin	Specify the key password source

An example of the signed certificate:

```
-----BEGIN CERTIFICATE-----
MIIDBTCCAe0CCQDBKbNGSE8C9DANBgkqhkiG9w0BAQsFADBCMQswCQYDVQQGEwJV
UzEMMAoGA1UEChMDTExQMRQwEgYDVQQLEwtUZXN0Q2x1c3RlcjEPMA0GA1UEAxMG
cm9vdENhMB4XDTE2MDkxMDA0MDAzMFoXDTE3MDkxMDA0MDAzMFowRzELMAkGA1UE
BhMCVVMxDDAKBgNVBAoTA0xMUDESMBAGA1UECxMJTExQMDkwNzE2MRYwFAYDVQQD
Ew0xMC4yMDAuMTc1LjE1MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA
hUJWFQ+9Xgs8KfhLYCBi8i0csY44YkCBFAz3jh6g80wdCqYetYcNxWi+3rgAFL6B
Aibt23xGqBKtNszxyUu8M7ocfw+dVeT4YQJWE0TNAnwlijx+jyl2IKCFtuON3NmG
vvHviPDyNyz2VuB7jlA3fGuCSsXEzpDXdStKZNcozeIxnmLRv2VKkp8edOX0Bi3Q
OxwKgwzZQ0/Z7yWaixl1WB2YKzY9s1BnoEoRHdFeQZR7f9QuQYBwIKsCx3cFsiEQ
JnpKqbZYLFrPmpMrR7ynLx6MGZVDuLfrXp/yFyuHmamIM80CGvKljwr/onalDY2F
5AeSdUyBM3vVTWrC9n9jwQIDAQABMA0GCSqGSIb3DQEBCwUAA4IBAQCqqXSKsKlW
sivk9/ap57fVbvYWj34FCmCYEWAPTrj0gEDwEP2WU2/813FF3fiXeFGwuNcm3XHl
0jsPsrckVmtko2ERGHCsQS7RlRlbRRzinQZ6zQaHFyDqsVBGeb/FRE0eJPO2OWQA
hksT1y7DAMv0kFyzvHDGtJRzWgXMpjc5LrWto46+JByx+9JjVI5a9DuKdvuoJGL/
CShFW/AWyOBk8LFlx+qzcYBy1R6WYqqE+pIhsq8X9Jtb6/ZymZBw7Ek9XH8ULNjV
S8QfiEkEXMjH+s+7Pofky7/8/udrEemQgLcIY3xKnlp+Rsz/wH21ZdKGs/lhbIzm
Xo+7pYd2dqHT
-----END CERTIFICATE-----
```

Verify the signed certificate for each node

7. Check the signed certificate by designating the rootCa certificate and the signed certificate to verify:

   ```
   openssl verify -CAfile rootCa.crt 10.200.175.15.crt_signed
   ```

   If the verification succeeds, a console message is returned:

   ```
   10.200.175.15.crt_signed: OK
   ```

Import rootCa certificate to each node keystore

8. Use `keytool -importcert` to import the rootCa certificate into each node keystore:

   ```
   keytool -importcert
   ```

```
    -keystore 10.200.175.15.jks
    -alias rootCa
    -file rootCa.crt
    -noprompt
    -keypass myKeyPass
    -storepass myKeyPass
```

The `-noprompt` option allows the command to use the specified values rather than prompting for input.

The keystore file now has two entries, one for the rootCa certificate and one for the node certificate:

```
Keystore type: JKS
Keystore provider: SUN

Your keystore contains 2 entries

rootca, Sep 10, 2016, trustedCertEntry,
Certificate fingerprint (SHA1):
 CD:F2:A5:6F:EC:DA:B9:9E:C1:8D:89:09:AE:FF:DB:BD:56:F6:7D:79
10.200.175.15, Sep 10, 2016, PrivateKeyEntry,
Certificate fingerprint (SHA1):
 D6:1B:6C:FE:E3:3B:4B:3D:E0:F0:38:EA:54:AD:F0:E7:1E:D4:CB:4D
```

Import node's signed certificate into node keystore for each node

9.

```
keytool -importcert
    -keystore 10.200.175.15.jks
    -alias 10.200.175.15
    -file 10.200.175.15.crt_signed
    -noprompt
    -keypass myKeyPass
    -storepass myKeyPass
```

The resulting file will appear similar to the result from the previous step, but the node certificate originally created is replaced with the signed node certificate.

Create a server truststore

10. A server truststore file can be used to establish a chain of trust between the nodes of the cluster.

```
keytool -importcert
    -keystore generic-server-truststore.jks
    -alias rootCa
    -file rootCa.crt
    -noprompt
    -keypass myPass
    -storepass truststorePass
```

The resulting truststore file can be inspected using the `keytool -list` command:

```
keytool -list
    -keystore generic-server-truststore.jks
    -storepass truststorePass
```

and an example of the truststore file will include a rootCa certificate entry:

```
Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

rootca, Sep 10, 2016, trustedCertEntry,
Certificate fingerprint (SHA1):
 CD:F2:A5:6F:EC:DA:B9:9E:C1:8D:89:09:AE:FF:DB:BD:56:F6:7D:79
```

Copy the truststore file to each node

11. The truststore file must be copied to each node. If a node is used to generate the file, copy the file to a location of choice and name the file with a standard format, such as `server-truststore.jks`. This example shows the copy command for a Linux server with a tarball installation of Cassandra and stores the file in the configuration directory of Cassandra:

    ```
    cp ~/generic-server-truststore.jks /usr/local/lib/cassandra/conf/server-truststore.jks
    ```

Copy the each node keystore file to each node

12. Each node file must have a copy of its keystore file. If a node is used to generate the file, copy the file to a location of choice. This example shows the secure remote copy commands for a Linux server with a tarball installation of Cassandra, where the certificates were generated on a single node. The file is stored in the configuration directory of Cassandra in this example:

    ```
    scp -r 10.200.175.150.jks /usr/local/lib/cassandra/conf/10.200.175.150.jks
    ```

Node-to-node encryption

Node-to-node encryption protects data transferred between nodes in a cluster, including gossip communications, using SSL (Secure Sockets Layer).

Prerequisites

Prepare SSL certificates with a self-signed CA for production, or prepare SSL certificates for development.

To enable node-to-node SSL, you must set the server_encryption_options in the cassandra.yaml file.

Procedure

Enable `server_encryption_options` on each node

1. Modify the cassandra.yaml file with the following settings:

    For production clusters:

    ```
    server_encryption_options:
        internode_encryption: all
        keystore: /usr/local/lib/cassandra/conf/server-keystore.jks
        keystore_password: myKeyPass
        truststore: /usr/local/lib/cassandra/conf/server-truststore.jks
        truststore_password: truststorePass
        # More advanced defaults below:
        protocol: TLS
        algorithm: SunX509
        store_type: JKS
        cipher_suites: [TLS_RSA_WITH_AES_256_CBC_SHA]
        require_client_auth: true
    ```

    This file uses the certificates prepared with a self-signed CA.

    For development clusters:

    ```
    server_encryption_options:
        internode_encryption: all
        keystore: /conf/keystore.node0
        keystore_password: cassandra
        truststore: /conf/truststore.node0
        truststore_password: cassandra
    ```

```
    # More advanced defaults below:
    protocol: TLS
    algorithm: SunX509
    store_type: JKS
    cipher_suites: [TLS_RSA_WITH_AES_256_CBC_SHA]
    require_client_auth: true
```

This file uses the certificates prepared for development.

Internode encryption can be set to four different choices:

all

All traffic is encrypted.

none

No traffic is encrypted.

dc

Traffic between datacenters is encrypted.

rack

Traffic between racks is encrypted.

Set appropriate paths to the `keystore` and `truststore` files. Set the passwords to the passwords set during keystore and truststore generation. If two-way certificate authentication is desired, set `require_client_auth` to `true`.

Restart cassandra

2. Restart cassandra to make changes effective.

```
$ kill -9 cassandra_pid
$ cassandra
```

3. Check the logs to discover if SSL encryption has been started. On Linux, use the `grep` command:

```
$ grep SSL install_location/logs/system.log
```

```
grep SSL %CASSANDRA_HOME%\logs\system.log
```

The resulting line will be similar to this example:

```
 INFO  [main] 2016-09-12 18:34:14,478 MessagingService.java:511 - Starting
  Encrypted Messaging Service on SSL port 7001
```

Client-to-node encryption

Client-to-node encryption protects data in flight from client machines to a database cluster using SSL (Secure Sockets Layer). It establishes a secure channel between the client and the coordinator node.

Prerequisites

Prepare SSL certificates with a self-signed CA for production, or prepare SSL certificates for development.

To enable client-to-node SSL, set the client_encryption_options in the cassandra.yaml file.

Procedure

On each node under client_encryption_options:

1. Enable encryption.

Enable `client_encryption_options` on each node

2. Modify the cassandra.yaml file with the following settings:

   For production clusters:

```
client_encryption_options:
    enabled: true
    # If enabled and optional is set to true encrypted and unencrypted
 connections are handled.
    optional: false
    keystore: /usr/local/lib/cassandra/conf/server-keystore.jks
    keystore_password: myKeyPass

    require_client_auth: true
    # Set trustore and truststore_password if require_client_auth is true
    truststore: /usr/local/lib/cassandra/conf/server-truststore.jks
    truststore_password: truststorePass
    protocol: TLS
    algorithm: SunX509
    store_type: JKS
    cipher_suites: [TLS_RSA_WITH_AES_256_CBC_SHA]
```

   This file uses the certificates prepared with a self-signed CA.

   For development clusters:

```
client_encryption_options:
    enabled: true
    # If enabled and optional is set to true encrypted and unencrypted
 connections are handled.
    optional: false
    keystore: conf/keystore.node0
    keystore_password: cassandra

    require_client_auth: true
    # Set trustore and truststore_password if require_client_auth is true
    truststore: conf/truststore.node0
    truststore_password: cassandra
    protocol: TLS
    algorithm: SunX509
    store_type: JKS
    cipher_suites: [TLS_RSA_WITH_AES_256_CBC_SHA]
```

   This file uses the certificates prepared for development.

   Set appropriate paths to the `keystore` and `truststore` files. Set the passwords to the passwords set during keystore and truststore generation. If two-way certificate authentication is desired, set `require_client_auth` to `true`. Enabling two-way certificate authentication allows tools to connect to a remote node. For local access to run `cqlsh` on a local node with SSL encryption, `require_client_auth` can be set to `false`

   Enabling client encryption will encrypt all traffic on the `native_transport_port` (default: 9042). If both encrypted and unencrypted traffic is required, an additional cassandra.yaml setting must be enabled. The `native_transport_port_ssl` (default: 9142) sets an additional dedicated port to carry encrypted transmissions, while `native_transport_port` carries unencrypted transmissions.

   Note: It is beneficial to install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files if this option is enabled.

Restart cassandra

3. Restart cassandra to make changes effective.

```
$ kill -9 cassandra_pid
$ cassandra
```

Using cqlsh with SSL

Using a `cqlshrc` file is the easiest method of getting `cqlshrc` settings. The cqlshrc.sample provides an example that can be copied as a starting point.

Prerequisites

Prepare SSL certificates with a self-signed CA for production, or prepare SSL certificates for development. Additionally, configure client-to-node encryption.

Procedure

1. To run cqlsh with SSL encryption, create a `.cassandra/cqlshrc` file in your home or client program directory with the following settings:

   For production clusters:

   ```
    [authentication]
   username = fred
   password = !!bang!!$

    [connection]
   hostname = 127.0.0.1
   port = 9042
   factory = cqlshlib.ssl.ssl_transport_factory

    [ssl]
   certfile = ~/.cassandra/rootCa.crt
   ;; Optional, true by default
   validate = true

   ;; To be provided when require_client_auth=true
   userkey = ~/.cassandra/rootCa.key

   ;; To be provided when require_client_auth=true
   usercert = ~/.cassandra/rootCa.crt
   ```

   This file uses the certificates prepared with a self-signed CA.

   For development clusters:

   ```
    [authentication]
   username = fred
   password = !!bang!!$

    [connection]
   hostname = 127.0.0.1
   port = 9042
   factory = cqlshlib.ssl.ssl_transport_factory

    [ssl]
   certfile = ~/keys/node0.cer.pem
   # Optional, true by default
   validate = true
   # The next 2 lines must be provided when require_client_auth = true in the
    cassandra.yaml file
   ```

```
userkey = ~/node0.key.pem
usercert = ~/node0.cer.pem

[certfiles]
# Optional section, overrides the default certfile in the [ssl] section
 for 2 way SSL
172.31.10.22 = ~/keys/node0.cer.pem
172.31.8.141 = ~/keys/node1.cer.pem
```

This file uses the certificates prepared for development. The use of the same IP addresses in the `[certfiles]` section, as is used to generate the *dname* of the certificates, is required for two-way SSL encryption. Each node must have a line in the `[certfiles]` section for client-to-remote-node or node-to-node.

When `validate` is enabled, to verify that the certificate is trusted the host in the certificate is compared to the host of the machine to which it is connected. Note that the rootCa certificate and key are supplied to access the trustchain. The SSL certificate must be provided either in the configuration file or as an environment variable. The environment variables (*SSL_CERTFILE* and *SSL_VALIDATE*) override any options set in this file.

Note: Additional settings in the `cqlshrc` file are described in Creating and using the cqlshrc file.

An optional section, `[certfiles]`, will override the default `certfile` in the `[ssl]` section. The use of the same IP addresses in the `[certfiles]` section, as is used to generate the `dname` of the certificates, is required for two-way SSL encryption. Each node must have a line in the `[certfiles]` section for client-to-remote-node or node-to-node. Using `certfiles]` is more common for development clusters.

2. Start cqlsh with the --ssl option for `cqlsh` to local node encrypted connection.

   ```
   $ cqlsh --ssl ## Package installations
   $ install_location/bin/cqlsh --ssl ## Tarball installations
   ```
3. A username and password can also be supplied at cqlsh startup. This example provides the username *cassandra* with password *cassandra*.

   ```
   $ cqlsh --ssl ## Package installations
   $ install_location/bin/cqlsh --ssl -u cassandra -p cassandra ## Tarball
    installations
   ```

   Note that a username and password can be entered in the `cqlshrc` file so that it will be automatically read each time `cqlsh` is started.
4. For a remote node encrypted connection, start cqlsh with the --ssl option and an IP address:

   ```
   $ cqlsh --ssl ## Package installations
   $ install_location/bin/cqlsh --ssl 172.31.10.22 ## Tarball installations
   ```

Using nodetool (JMX) with SSL encryption

Using `nodetool` with SSL requires some JMX setup. Changes to cassandra-env.sh are required, and a configuration file, `~/.cassandra/nodetool-ssl.properties`, is created.

Prerequisites

Prepare SSL certificates with a self-signed CA for production, or prepare SSL certificates for development. Additionally, configure client-to-node encryption.

Procedure

1. First, follow steps #1-8 in Enabling JMX authentication and authorization on page 101 if authentication and authorization are required.
2. To run `nodetool` with SSL encryption, some additional changes are required to cassandra-env.sh. The following settings must be added to the file. Use the file path to the keystore and truststore, and appropriate passwords for each file. These changes must be made on each node in the cluster.

   For production:

   ```
   JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.ssl=true"
     JVM_OPTS="$JVM_OPTS -
   Dcom.sun.management.jmxremote.ssl.need.client.auth=true"
     JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.registry.ssl=true"
     #JVM_OPTS="$JVM_OPTS -
   Dcom.sun.management.jmxremote.ssl.enabled.protocols=<enabled-protocols>"
     #JVM_OPTS="$JVM_OPTS -
   Dcom.sun.management.jmxremote.ssl.enabled.cipher.suites=<enabled-cipher-
   suites>"

     JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.keyStore=/usr/local/lib/cassandra/
   conf/server-keystore.jks"
     JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.keyStorePassword=myKeyPass"
     JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.trustStore=/usr/local/lib/cassandra/
   conf/server-truststore.jks"
     JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.trustStorePassword=truststorePass"
   ```

   For development:

   ```
      JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.ssl=true"
     JVM_OPTS="$JVM_OPTS -
   Dcom.sun.management.jmxremote.ssl.need.client.auth=true"
     JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.registry.ssl=true"
     #JVM_OPTS="$JVM_OPTS -
   Dcom.sun.management.jmxremote.ssl.enabled.protocols=<enabled-protocols>"
     #JVM_OPTS="$JVM_OPTS -
   Dcom.sun.management.jmxremote.ssl.enabled.cipher.suites=<enabled-cipher-
   suites>"

     JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.keyStore=keystore.node0"
     JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.keyStorePassword=cassandra"
     JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.trustStore=truststore.node0"
     JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.trustStorePassword=cassandra"
   ```

   Enable SSL for JMX by setting `com.sun.management.jmxremote.ssl` to `true`. If two-way certificate authentication is desired, set `com.sun.management.jmxremote.ssl.need.client.auth` to `true`. If `com.sun.management.jmxremote.registry.ssl` is set to `true`, an RMI registry protected by SSL will be created and configured by the out-of-the-box management agent when the Java VM is started. If the `com.sun.management.jmxremote.registry.ssl` property is set to `true`, to have full security then `com.sun.management.jmxremote.ssl.need.client.auth` must also be enabled. Set appropriate paths to the `keystore` and `truststore` files. Set the passwords to the passwords set during keystore and truststore generation.
3. Restart Cassandra.
4. To run `nodetool` with SSL encryption, create a `.cassandra/nodetool-ssl.properties` file in your home or client program directory with the following settings on the node on which `nodetool` will run.

For production:

```
-Dcom.sun.management.jmxremote.ssl=true
-Dcom.sun.management.jmxremote.ssl.need.client.auth=true
-Dcom.sun.management.jmxremote.registry.ssl=true
-Djavax.net.ssl.keyStore=/usr/local/lib/cassandra/conf/server-keystore.jks
-Djavax.net.ssl.keyStorePassword=myKeyPass
-Djavax.net.ssl.trustStore=/usr/local/lib/cassandra/conf/server-
truststore.jks
-Djavax.net.ssl.trustStorePassword=truststorePass
```

For development:

```
-Djavax.net.ssl.keyStore=keystore.node0
-Djavax.net.ssl.keyStorePassword=cassandra
-Djavax.net.ssl.trustStore=truststore.node0
-Djavax.net.ssl.trustStorePassword=cassandra
-Dcom.sun.management.jmxremote.ssl.need.client.auth=true
-Dcom.sun.management.jmxremote.registry.ssl=true
```

5. Start `nodetool` with the --ssl option for encrypted connection for any `nodetool` operation.

```
$ nodetool --ssl info ## Package installations
$ install_location/bin/nodetool -ssl info ## Tarball installations
```

6. Start `nodetool` with the --ssl option for encrypted connection and a username and password for authentication and authorization for any `nodetool` operation.

```
$ nodetool --ssl -u cassandra -pw cassandra status ## Package installations
$ install_location/bin/nodetool -ssl -u cassandra -pw cassandra status ##
 Tarball installations
```

Using jconsole (JMX) with SSL encryption

Using `jconsole` with SSL requires the same JMX changes to cassandra-env.sh as `nodetool`. See using nodetool (JMX) with SSL encryption. There is no need to create `nodetool-ssl.properties`, but the same JVM keystore and truststore options must be specified with `jconsole` on the command line.

Prerequisites

Prepare SSL certificates with a self-signed CA for production, or prepare SSL certificates for development. Additionally, configure client-to-node encryption.

Procedure

1. Copy the keystore and truststore files created in the prerequisite to the node where jconsole will be run. In this example, the files are `server-keystore.jks` and `server-truststore.jks`.
2. Run `jconsole` using the JVM options:

```
jconsole -J-Djavax.net.ssl.keyStore=server-keystore.jks
-J-Djavax.net.ssl.keyStorePassword=myKeyPass
-J-Djavax.net.ssl.trustStore=server-truststore.jks
-J-Djavax.net.ssl.trustStorePassword=truststorePass
```

   If no errors occur, `jconsole` will start. If connecting to a remote node, enter the hostname and JMX port, in *Remote Process*. If using authentication, enter the username and password.

Configuring firewall port access

The following ports must be open to allow bi-directional communication between nodes, including certain Cassandra ports. Configure the firewall running on nodes in your Cassandra cluster accordingly. Without open ports as shown, nodes will act as a standalone database server and will not join the Cassandra cluster.

Table: Public port

Port number.	Description
22	SSH port

Table: Cassandra inter-node ports

Port number.	Description
7000	Cassandra inter-node cluster communication.
7001	Cassandra SSL inter-node cluster communication.
7199	Cassandra JMX monitoring port.

Table: Cassandra client ports

Port number.	Description
9042	Cassandra client port.
9160	Cassandra client port (Thrift).
9142	Default for native_transport_port_ssl, useful when both encrypted and unencrypted connections a

Configuring gossip settings

When a node first starts up, it looks at its cassandra.yaml configuration file to determine the name of the Cassandra cluster it belongs to; which nodes (called *seeds*) to contact to obtain information about the other nodes in the cluster; and other parameters for determining port and range information.

Procedure

In the `cassandra.yaml` file, set the following parameters:

Property	Description
cluster_name	Name of the cluster that this node is joining. Must be the same for every node in the cluster.
listen_address	The IP address or hostname that Cassandra binds to for connecting this node to other nodes.
listen_interface	Use this option instead of listen_address to specify the network interface by name, rather than address/hostname
(Optional) broadcast_address	The "public" IP address this node uses to broadcast to other nodes outside the network or across regions in multiple-region EC2 deployments. If this property is commented out, the node uses the same IP address or hostname as `listen_address`.

Property	Description
	A node does not need a separate `broadcast_address` in a single-node or single-datacenter installation, or in an EC2-based network that supports automatic switching between private and public communication. It is necessary to set a separate `listen_address` and `broadcast_address` on a node with multiple physical network interfaces or other topologies where not all nodes have access to other nodes by their private IP addresses. For specific configurations, see the instructions for listen_address. The default is the listen_address.
seed_provider	A -seeds list is comma-delimited list of hosts (IP addresses) that gossip uses to learn the topology of the ring. Every node should have the same list of seeds. **Attention:** In multiple data-center clusters, include at least one node from each datacenter (replication group) in the seed list. Designating more than a single seed node per datacenter is recommended for fault tolerance. Otherwise, gossip has to communicate with another datacenter when bootstrapping a node. Making every node a seed node is **not** recommended because of increased maintenance and reduced gossip performance. Gossip optimization is not critical, but it is recommended to use a small seed list (approximately three nodes per datacenter).
storage_port	The inter-node communication port (default is 7000). Must be the same for every node in the cluster.
initial_token	For legacy clusters. Set this property for single-node-per-token architecture, in which a node owns exactly one contiguous range in the ring space.
num_tokens	For new clusters. The number of tokens randomly assigned to this node in a cluster that uses virtual nodes (vnodes).

Configuring the heap dump directory

Analyzing the heap dump file can help troubleshoot memory problems. Cassandra starts Java with the option `-XX:+HeapDumpOnOutOfMemoryError`. Using this option triggers a heap dump in the event of an out-of-memory condition. The heap dump file consists of references to objects that cause the heap to overflow. By default, Cassandra puts the file a subdirectory of the working, root directory when running as a service. If Cassandra does not have write permission to the root directory, the heap dump fails. If the root directory is too small to accommodate the heap dump, the server crashes.

To ensure that a heap dump succeeds and to prevent crashes, configure a heap dump directory that is:

- Accessible to Cassandra for writing
- Large enough to accommodate a heap dump

Base the size of the directory on the value of the Java `-mx` option.

Procedure

Set the location of the heap dump in the cassandra-env.sh file.

1. Open the `cassandra-env.sh` file for editing.

2. Scroll down to the comment about the heap dump path:

   ```
   set jvm HeapDumpPath with CASSANDRA_HEAPDUMP_DIR
   ```

3. On the line after the comment, set the `CASSANDRA_HEAPDUMP_DIR` to the path you want to use:

   ```
   set jvm HeapDumpPath with CASSANDRA_HEAPDUMP_DIR CASSANDRA_HEAPDUMP_DIR
    =<path>
   ```

4. Save the `cassandra-env.sh` file and restart.

Configuring virtual nodes

Enabling virtual nodes on a new cluster

Generally when all nodes have equal hardware capability, they should have the same number of virtual nodes (vnodes). If the hardware capabilities vary among the nodes in your cluster, assign a proportional number of vnodes to the larger machines. For example, you could designate your older machines to use 128 vnodes and your new machines (that are twice as powerful) with 256 vnodes.

Procedure

Set the number of tokens on each node in your cluster with the num_tokens parameter in the cassandra.yaml file.

The recommended value is 256. Do not set the initial_token parameter.

Related information

Install locations on page 71

Enabling virtual nodes on an existing production cluster

You cannot directly convert a single-token nodes to a vnode. However, you can configure another datacenter configured with vnodes already enabled and let Cassandra automatic mechanisms distribute the existing data into the new nodes. This method has the least impact on performance.

Procedure

1. Add a new datacenter to the cluster.
2. Once the new datacenter with vnodes enabled is up, switch your clients to use the new datacenter.
3. Run a full repair with nodetool repair.

   This step ensures that after you move the client to the new datacenter that any previous writes are added to the new datacenter and that nothing else, such as hints, is dropped when you remove the old datacenter.
4. Update your schema to no longer reference the old datacenter.
5. Remove the old datacenter from the cluster.

   See Decommissioning a datacenter on page 146.

Using multiple network interfaces

How to configure Cassandra for multiple network interfaces or when using different regions in cloud implementations.

You must configure settings in both the `cassandra.yaml` file and the property file (`cassandra-rackdc.properties` or `cassandra-topology.properties`) used by the snitch.

Configuring cassandra.yaml for multiple networks or across regions in cloud implementations

In multiple networks or cross-region cloud scenarios, communication between datacenters can only take place using an external IP address. The external IP address is defined in the `cassandra.yaml` file using the broadcast_address setting. Configure each node as follows:

1. In the cassandra.yaml, set the listen_address to the *private* IP address of the node, and the broadcast_address to the *public* address of the node.

   This allows Cassandra nodes to bind to nodes in another network or region, thus enabling multiple data-center support. For intra-network or region traffic, Cassandra switches to the private IP after establishing a connection.
2. Set the addresses of the seed nodes in the `cassandra.yaml` file to that of the *public* IP. Private IP are not routable between networks. For example:

   ```
   seeds: 50.34.16.33, 60.247.70.52
   ```

   Note: Do not make all nodes seeds, see Internode communications (gossip) on page 13.
3. Be sure that the storage_port or ssl_storage_port is open on the public IP firewall.

CAUTION: Be sure to enable encryption and authentication when using public IPs. See Node-to-node encryption on page 115. Another option is to use a custom VPN to have local, inter-region/ datacenter IPs.

Additional cassandra.yaml configuration for non-EC2 implementations

If multiple network interfaces are used in a non-EC2 implementation, enable the `listen_on_broadcast_address` option.

```
listen_on_broadcast_address: true
```

In non-EC2 environments, the public address to private address routing is not automatically enabled. Enabling `listen_on_broadcast_address` allows Cassandra to listen on both `listen_address` and `broadcast_address` with two network interfaces.

Configuring the snitch for multiple networks

External communication between the datacenters can only happen when using the broadcast_address (public IP).

The GossipingPropertyFileSnitch on page 22 is recommended for production. The `cassandra-rackdc.properties` file defines the datacenters used by this snitch. Enable the option `prefer_local` to ensure that traffic to `broadcast_address` will re-route to `listen_address`.

For each node in the network, specify its datacenter in cassandra-rackdc.properties file.

In the example below, there are two cassandra datacenters and each datacenter is named for its workload. The datacenter naming convention in this example is based on the workload. You can use other conventions, such as DC1, DC2 or 100, 200. (datacenter names are case-sensitive.)

Network A	Network B

Node and datacenter:
• **node0**
dc=DC_A_cassandra
rack=RAC1
• **node1**
dc=DC_A_cassandra
rack=RAC1
• **node2**
dc=DC_B_cassandra
rack=RAC1
• **node3**
dc=DC_B_cassandra
rack=RAC1
• **node4**
dc=DC_A_analytics
rack=RAC1
• **node5**
dc=DC_A_search
rack=RAC1
	Node and datacenter:
	• **node0**
	dc=DC_A_cassandra
	rack=RAC1
	• **node1**
	dc=DC_A_cassandra
	rack=RAC1
	• **node2**
	dc=DC_B_cassandra
	rack=RAC1
	• **node3**
	dc=DC_B_cassandra
	rack=RAC1
	• **node4**
	dc=DC_A_analytics
	rack=RAC1
	• **node5**
	dc=DC_A_search
	rack=RAC1

Configuring the snitch for cross-region communication in cloud implementations

Note: Be sure to use the appropriate snitch for your implementation. If your deploying on Amazon EC2, see the instructions in Ec2MultiRegionSnitch on page 23.

In cloud deployments, the region name is treated as the datacenter name and availability zones are treated as racks within a datacenter. For example, if a node is in the us-east-1 region, us-east is the datacenter name and 1 is the rack location. (Racks are important for distributing replicas, but not for datacenter naming.)

In the example below, there are two cassandra datacenters and each datacenter is named for its workload. The datacenter naming convention in this example is based on the workload. You can use other conventions, such as DC1, DC2 or 100, 200. (datacenter names are case-sensitive.)

For each node, specify its datacenter in the cassandra-rackdc.properties. The dc_suffix option defines the datacenters used by the snitch. Any other lines are ignored.

Region: us-east	Region: us-west
Node and datacenter: • **node0** `dc_suffix=_1_cassandra` • **node1** `dc_suffix=_1_cassandra` • **node2**	Node and datacenter: • **node0** `dc_suffix=_1_cassandra` • **node1** `dc_suffix=_1_cassandra` • **node2**

Region: us-east	Region: us-west

`dc_suffix=_2_cassandra`
• **node3**
`dc_suffix=_2_cassandra`
• **node4**
`dc_suffix=_1_analytics`
• **node5**
`dc_suffix=_1_search`
This results in four us-east datacenters:
`us-east_1_cassandra`
`us-east_2_cassandra`
`us-east_1_analytics`
`us-east_1_search`
	`dc_suffix=_2_cassandra`
	• **node3**
	`dc_suffix=_2_cassandra`
	• **node4**
	`dc_suffix=_1_analytics`
	• **node5**
	`dc_suffix=_1_search`
	This results in four us-west datacenters:
	`us-west_1_cassandra`
	`us-west_2_cassandra`
	`us-west_1_analytics`
	`us-west_1_search`

Configuring logging

Cassandra provides logging functionality using Simple Logging Facade for Java (SLF4J) with a logback backend. Logs are written to the `system.log` and `debug.log`in the Cassandra logging directory. You can configure logging programmatically or manually. Manual ways to configure logging are:

- Run the `nodetool setlogginglevel` command.
- Configure the `logback-test.xml` or `logback.xml` file installed with Cassandra.
- Use the JConsole tool to configure logging through JMX.

Logback looks for `logback-test.xml` first, and then for logback.xml file.

The XML configuration files looks like this:

```
<configuration scan="true">
  <jmxConfigurator />
  <appender name="FILE"
 class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>${cassandra.logdir}/system.log</file>
    <rollingPolicy
 class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
      <fileNamePattern>${cassandra.logdir}/system.log.%i.zip</
fileNamePattern>
      <minIndex>1</minIndex>
      <maxIndex>20</maxIndex>
    </rollingPolicy>

    <triggeringPolicy
 class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
      <maxFileSize>20MB</maxFileSize>
    </triggeringPolicy>
    <encoder>
      <pattern>%-5level [%thread] %date{ISO8601} %F:%L - %msg%n</pattern>
      <!-- old-style log format
      <pattern>%5level [%thread] %date{ISO8601} %F (line %L) %msg%n</
pattern>
      -->
    </encoder>
  </appender>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%-5level %date{HH:mm:ss,SSS} %msg%n</pattern>
    </encoder>
  </appender>

  <root level="INFO">
    <appender-ref ref="FILE" />
    <appender-ref ref="STDOUT" />
  </root>

  <logger name="com.thinkaurelius.thrift" level="ERROR"/>
</configuration>
```

The appender configurations specify where to print the log and its configuration. The first appender directs logs to a file. The second appender directs logs to the console. You can change the following logging functionality:

- Rolling policy
  - The policy for rolling logs over to an archive
  - Location and name of the log file
  - Location and name of the archive
  - Minimum and maximum file size to trigger rolling
- Format of the message
- The log level

Log levels

The valid values for setting the log level include ALL for logging information at all levels, TRACE through ERROR, and OFF for no logging. TRACE creates the most verbose log, and ERROR, the least.

- ALL
- TRACE
- DEBUG
- INFO (Default)
- WARN
- ERROR
- OFF

Note: Increasing logging levels can generate heavy logging output on a moderately trafficked cluster.

You can use the `nodetool getlogginglevels` command to see the current logging configuration.

```
$ nodetool getlogginglevels
Logger Name                                        Log Level
ROOT                                               INFO
com.thinkaurelius.thrift                           ERROR
```

To add debug logging to a class permanently using the logback framework, use `nodetool setlogginglevel` to check you have the right class before you set it in the logback.xml file in `install_location/conf`. Modify to include the following line or similar at the end of the file:

```
<logger name="org.apache.cassandra.gms.FailureDetector" level="DEBUG"/>
```

Restart the node to invoke the change.

Migrating to logback from log4j

If you upgrade from a previous version of Cassandra that used log4j, you can convert *log4j.properties* files to *logback.xml* using the logback PropertiesTranslator web-application.

Using log file rotation

The default policy rolls the `system.log` file after the size exceeds 20MB. Archives are compressed in zip format. Logback names the log files `system.log.1.zip`, `system.log.2.zip`, and so on. For more information, see logback documentation.

Enabling extended compaction logging

You can configure Casandra to collect in-depth information about compaction activity on a node, and write it to a dedicated log file. For details, see Enabling extended compaction logging.

Commit log archive configuration

Cassandra provides commit log archiving and point-in-time recovery. The commit log is archived at node startup and when a commit log is written to disk, or at a specified point-in-time. You configure this feature in the commitlog_archiving.properties configuration file.

The commands `archive_command` and `restore_command` expect only a single command with arguments. The parameters must be entered verbatim. STDOUT and STDIN or multiple commands cannot be executed. To workaround, you can script multiple commands and add a pointer to this file. To disable a command, leave it blank.

Procedure

- Archive a commit log segment:

Command	**archive_command=**	
Parameters	`%path`	Fully qualified path of the segment to archive.
	`%name`	Name of the commit log.
Example	`archive_command=/bin/ln %path /backup/%name`	

- Restore an archived commit log:

Command	**restore_command=**	
Parameters	`%from`	Fully qualified path of the an archived commitlog segment fr
	`%to`	Name of live commit log directory.
Example	`restore_command=cp -f %from %to`	

- Set the restore directory location:

Command	**restore_directories=**
Format	`restore_directories=`*`restore_directory_location`*

- Restore mutations created up to and including the specified timestamp:

Command	**restore_point_in_time=**
Format	`<timestamp>` (YYYY:MM:DD HH:MM:SS)

Command	restore_point_in_time=
Example	`restore_point_in_time=2013:12:11 17:00:00`

Restore stops when the first client-supplied timestamp is greater than the restore point timestamp. Because the order in which Cassandra receives mutations does not strictly follow the timestamp order, this can leave some mutations unrecovered.

Generating tokens

If not using virtual nodes (vnodes), you must calculate tokens for your cluster.

The following topics in the Cassandra 1.1 documentation provide conceptual information about tokens:

- Data Distribution in the Ring
- Replication Strategy

About calculating tokens for single or multiple datacenters in Cassandra 1.2 and later

- Single datacenter deployments: calculate tokens by dividing the hash range by the number of nodes in the cluster.
- Multiple datacenter deployments: calculate the tokens for each datacenter so that the hash range is evenly divided for the nodes in each datacenter.

For more explanation, see be sure to read the conceptual information mentioned above.

The method used for calculating tokens depends on the type of partitioner:

Calculating tokens for the Murmur3Partitioner

Use this method for generating tokens when you are **not** using virtual nodes (vnodes) and using the Murmur3Partitioner (default). This partitioner uses a maximum possible range of hash values from -2^{63} to $+2^{63}$-1. To calculate tokens for this partitioner:

```
$  python -c "print [str(((2**64 / number_of_tokens) * i) - 2**63) for i in
 range(number_of_tokens)]"
```

For example, to generate tokens for 6 nodes:

```
$  python -c "print [str(((2**64 / 6) * i) - 2**63) for i in range(6)]"
```

The command displays the token for each node:

```
[ '-9223372036854775808', '-6148914691236517206', '-3074457345618258604',
  '-2', '3074457345618258600', '6148914691236517202' ]
```

Calculating tokens for the RandomPartitioner

To calculate tokens when using the RandomPartitioner in Cassandra 1.2 clusters, use the Cassandra 1.1 Token Generating Tool.

Hadoop support

Cassandra support for integrating Hadoop with Cassandra includes:

- MapReduce

Notice: Apache Pig is no longer supported as of Cassandra 3.0.

You can use Cassandra 3.0 with Hadoop 2.x or 1.x with some restrictions:

- You must run separate datacenters: one or more datacenters with nodes running just Cassandra (for Online Transaction Processing) and others with nodes running C* & with Hadoop installed. See Isolate Cassandra and Hadoop for details.
- Before starting the datacenters of Cassandra/Hadoop nodes, disable virtual nodes (vnodes).

  Note: You only need to disable vnodes in datacenters with nodes running Cassandra AND Hadoop.

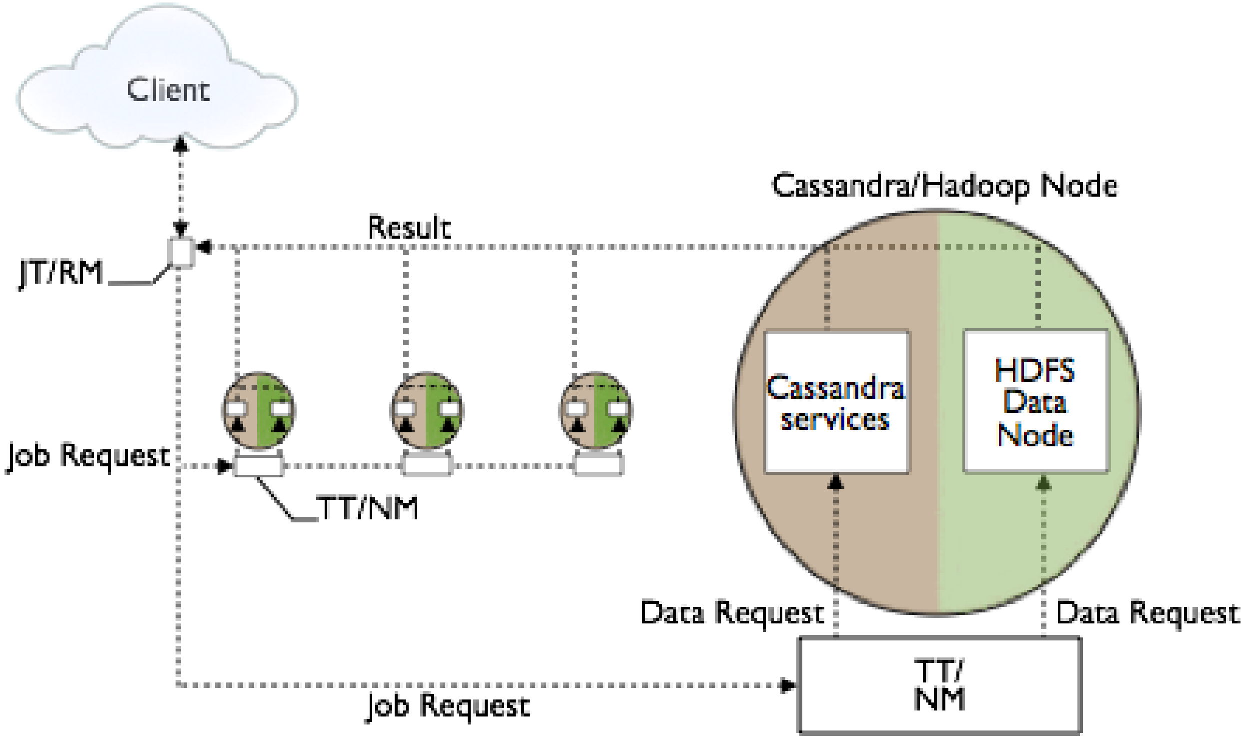
To disable virtual nodes:

1. In the cassandra.yaml file, set num_tokens to 1.
2. Uncomment the initial_token property and set it to 1 or to the value of a generated token for a multi-node cluster.
3. Start the cluster for the first time.

   You cannot convert single-token nodes to vnodes. See Enabling virtual nodes on an existing production clusterfor another option.

Setup and configuration, described in the Apache docs, involves overlaying a Hadoop cluster on Cassandra nodes, configuring a separate server for the Hadoop NameNode/JobTracker, and installing a Hadoop TaskTracker and Data Node on each Cassandra node. The nodes in the Cassandra datacenter can draw from data in the HDFS Data Node as well as from Cassandra. The Job Tracker/Resource Manager (JT/RM) receives MapReduce input from the client application. The JT/RM sends a MapReduce job request to the Task Trackers/Node Managers (TT/NM) and an optional clients MapReduce. The data is written to Cassandra and results sent back to the client.

MapReduce Process in a Cassandra/Hadoop Cluster



The Apache docs also cover how to get configuration and integration support.

Input and Output Formats

Hadoop jobs can receive data from CQL tables and indexes and can write their output to Cassandra tables as well as to the Hadoop FileSystem. Cassandra 3.0 supports the following formats for these tasks:

- `CqlInputFormat` class: for importing job input into the Hadoop filesystem from CQL tables
- `CqlOutputFormat` class: for writing job output from the Hadoop filesystem to CQL tables
- `CqlBulkOutputFormat` class: generates Cassandra SSTables from the output of Hadoop jobs, then loads them into the cluster using the `SSTableLoaderBulkOutputFormat` class

Reduce tasks can store keys (and corresponding bound variable values) as CQL rows (and respective columns) in a given CQL table.

Running the wordcount example

Wordcount example JARs are located in the `examples` directory of the Cassandra source code installation. There are CQL and legacy examples in the `hadoop_cql3_word_count` and `hadoop_word_count` subdirectories, respectively. Follow instructions in the readme files.

Isolating Hadoop and Cassandra workloads

When you create a keyspace using CQL, Cassandra creates a virtual datacenter for a cluster, even a one-node cluster, automatically. You assign nodes that run the same type of workload to the same datacenter. The separate, virtual datacenters for different types of nodes segregate workloads running Hadoop from those running Cassandra. Segregating workloads ensures that only one type of workload is active per datacenter. Separating nodes running a sequential data load, from nodes running any other type of workload, such as Cassandra real-time OLTP queries is a best practice.

Initializing a cluster

Initializing a multiple node cluster (single datacenter)

This topic contains information for deploying a Cassandra cluster with a single datacenter. If you're new to Cassandra, and haven't set up a cluster, see Planning and testing cluster deployments.

Prerequisites

Each node must be correctly configured before starting the cluster. You must determine or perform the following before starting the cluster:

- A good understanding of how Cassandra works. At minimum, be sure to read Understanding the architecture on page 11, especially the Data replication section, and Cassandra's rack feature.
- Install Cassandra on each node.
- Choose a name for the cluster.
- Get the IP address of each node.
- Determine which nodes will be seed nodes. **Do not make all nodes seed nodes.** Please read Internode communications (gossip) on page 13.
- Determine the snitch and replication strategy. The GossipingPropertyFileSnitch on page 22 and NetworkTopologyStrategy are recommended for production environments.
- Determine a naming convention for each rack. For example, good names are RAC1, RAC2 or R101, R102.

- The cassandra.yaml configuration file, and property files such as `cassandra-rackdc.properties`, give you more configuration options. See the Configuration section for more information.

This example describes installing a 6 node cluster spanning 2 racks in a single datacenter. Each node is already configured to use the GossipingPropertyFileSnitch and 256 virtual nodes (vnodes).

In Cassandra, "datacenter" is synonymous with "replication group". Both terms refer to a set of nodes configured as a group for replication purposes.

Procedure

1. Suppose you install Cassandra on these nodes:

   ```
   node0 110.82.155.0 (seed1)
   node1 110.82.155.1
   node2 110.82.155.2
   node3 110.82.156.3 (seed2)
   node4 110.82.156.4
   node5 110.82.156.5
   ```

   Note: It is a best practice to have more than one seed node per datacenter.
2. If you have a firewall running in your cluster, you must open certain ports for communication between the nodes. See Configuring firewall port access on page 122.
3. If Cassandra is running, you must stop the server and clear the data:

   Doing this removes the default cluster_name (Test Cluster) from the system table. All nodes must use the same cluster name.

   Package installations:

   a) Stop Cassandra:

      ```
      $ sudo service cassandra stop
      ```
   b) Clear the data:

      ```
      $ sudo rm -rf /var/lib/cassandra/data/system/*
      ```

   Tarball installations:

   a) Stop Cassandra:

      ```
      $ ps auwx | grep cassandra
      $ sudo  kill pid
      ```
   b) Clear the data:

      ```
      $ sudo rm -rf /var/lib/cassandra/data/data/system/*
      ```
4. Set the properties in the cassandra.yaml file for each node:

   Note: After making any changes in the `cassandra.yaml` file, you must restart the node for the changes to take effect.

   Properties to set:

   - cluster_name:
   - num_tokens: *recommended value: 256*
   - -seeds: *internal IP address of each seed node*

     In new clusters. Seed nodes don't perform bootstrap (the process of a new node joining an existing cluster.)
   - listen_address:

  If the node is a seed node, this address must match an IP address in the seeds list. Otherwise, gossip communication fails because it doesn't know that it is a seed.

  If not set, Cassandra asks the system for the local address, the one associated with its hostname. In some cases Cassandra doesn't produce the correct address and you must specify the listen_address.

- rpc_address:*listen address for client connections*
- endpoint_snitch: *name of snitch* (See endpoint_snitch.) If you are changing snitches, see Switching snitches on page 147.
- auto_bootstrap: false (Add this setting **only** when initializing a fresh cluster with no data.)

Note: If the nodes in the cluster are identical in terms of disk layout, shared libraries, and so on, you can use the same `cassandra.yaml` file on all of them.

Example:

```
cluster_name: 'MyCassandraCluster'
num_tokens: 256
seed_provider:
  - class_name: org.apache.cassandra.locator.SimpleSeedProvider
    parameters:
         - seeds: "110.82.155.0,110.82.155.3"
listen_address:
rpc_address: 0.0.0.0
endpoint_snitch: GossipingPropertyFileSnitch
```

If rpc_address is set to a wildcard address (`0.0.0.0`), then broadcast_rpc_address must be set, or the service won't even start.

5. In the `cassandra-rackdc.properties` file, assign the datacenter and rack names you determined in the Prerequisites. For example:

```
# indicate the rack and dc for this node
dc=DC1
rack=RAC1
```

6. The `GossipingPropertyFileSnitch` always loads `cassandra-topology.properties` when that file is present. Remove the file from each node on any new cluster or any cluster migrated from the `PropertyFileSnitch`.
7. After you have installed and configured Cassandra on all nodes, DataStax recommends starting the seed nodes one at a time, and then starting the rest of the nodes.

   Note: If the node has restarted because of automatic restart, you must first stop the node and clear the data directories, as described above.

   Package installations:

```
$ sudo service cassandra start
```

   Tarball installations:

```
$ cd install_location
$ bin/cassandra
```

8. To check that the ring is up and running, run:

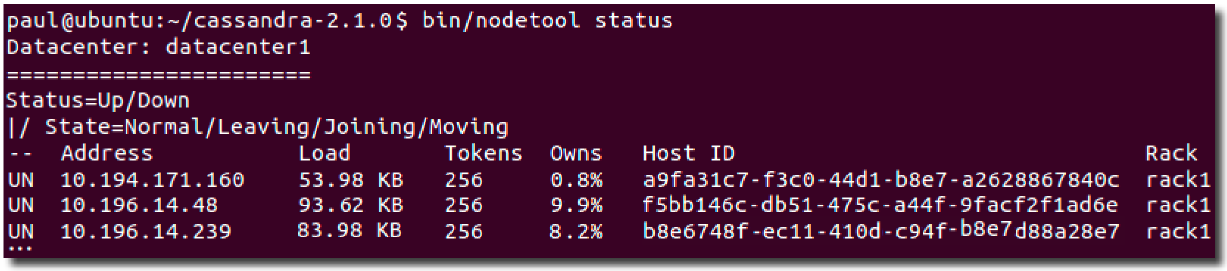
   Package installations:

```
$ nodetool status
```

   Tarball installations:

```
$ cd install_location
$ bin/nodetool status
```

The output should list each node, and show its status as `UN` (Up Normal).



```
paul@ubuntu:~/cassandra-2.1.0$ bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address          Load       Tokens  Owns   Host ID                               Rack
UN  10.194.171.160   53.98 KB   256     0.8%   a9fa31c7-f3c0-44d1-b8e7-a2628867840c  rack1
UN  10.196.14.48     93.62 KB   256     9.9%   f5bb146c-db51-475c-a44f-9facf2f1ad6e  rack1
UN  10.196.14.239    83.98 KB   256     8.2%   b8e6748f-ec11-410d-c94f-b8e7d88a28e7  rack1
...
```

Related information

Install locations on page 71

Initializing a multiple node cluster (multiple datacenters)

This topic contains information for deploying a Cassandra cluster with multiple datacenters. If you're new to Cassandra, and haven't set up a cluster, see Planning and testing cluster deployments.

This example describes installing a six node cluster spanning two datacenters. Each node is configured to use the GossipingPropertyFileSnitch (multiple rack aware) and 256 virtual nodes (vnodes).

In Cassandra, "datacenter" is synonymous with "replication group". Both terms refer to a set of nodes configured as a group for replication purposes.

Prerequisites

Each node must be correctly configured before starting the cluster. You must determine or perform the following before starting the cluster:

- A good understanding of how Cassandra works. At minimum, be sure to read Understanding the architecture on page 11 (especially the Data replication on page 17 section) and the rack feature of Cassandra.
- Install Cassandra on each node.
- Choose a name for the cluster.
- Get the IP address of each node.
- Determine which nodes will be seed nodes. **Do not make all nodes seed nodes.** Please read Internode communications (gossip) on page 13.
- Determine the snitch and replication strategy. The GossipingPropertyFileSnitch on page 22 and NetworkTopologyStrategy are recommended for production environments.
- Determine a naming convention for each datacenter and rack. Examples: DC1, DC2 or 100, 200 / RAC1, RAC2 or R101, R102. Choose the name carefully; renaming a datacenter is not possible.
- The cassandra.yaml configuration file, and property files such as `cassandra-rackdc.properties`, give you more configuration options. See the Configuration section for more information.

Procedure

1. Suppose you install Cassandra on these nodes:

```
node0 10.168.66.41 (seed1)
```

```
node1 10.176.43.66
node2 10.168.247.41
node3 10.176.170.59 (seed2)
node4 10.169.61.170
node5 10.169.30.138
```

Note: It is a best practice to have more than one seed node per datacenter.

2. If you have a firewall running in your cluster, you must open certain ports for communication between the nodes. See Configuring firewall port access on page 122.
3. If Cassandra is running, you must stop the server and clear the data:

   Doing this removes the default cluster_name (Test Cluster) from the system table. All nodes must use the same cluster name.

   Package installations:

   a) Stop Cassandra:

   ```
   $ sudo service cassandra stop
   ```

   b) Clear the data:

   ```
   $ sudo rm -rf /var/lib/cassandra/data/system/*
   ```

   Tarball installations:

   a) Stop Cassandra:

   ```
   $ ps auwx | grep cassandra
   $ sudo  kill pid
   ```

   b) Clear the data:

   ```
   $ sudo rm -rf /var/lib/cassandra/data/data/system/*
   ```

4. Set the properties in the cassandra.yaml file for each node:

   Note: After making any changes in the `cassandra.yaml` file, you must restart the node for the changes to take effect.

   Properties to set:

   - cluster_name:
   - num_tokens: *recommended value: 256*
   - -seeds: *internal IP address of each seed node*

     In new clusters. Seed nodes don't perform bootstrap (the process of a new node joining an existing cluster.)
   - listen_address:

     If the node is a seed node, this address must match an IP address in the seeds list. Otherwise, gossip communication fails because it doesn't know that it is a seed.

     If not set, Cassandra asks the system for the local address, the one associated with its hostname. In some cases Cassandra doesn't produce the correct address and you must specify the listen_address.
   - rpc_address:*listen address for client connections*
   - endpoint_snitch: *name of snitch* (See endpoint_snitch.) If you are changing snitches, see Switching snitches on page 147.
   - auto_bootstrap: false (Add this setting **only** when initializing a fresh cluster with no data.)

   Note: If the nodes in the cluster are identical in terms of disk layout, shared libraries, and so on, you can use the same `cassandra.yaml` file on all of them.

Example:

```
cluster_name: 'MyCassandraCluster'
num_tokens: 256
seed_provider:
  - class_name: org.apache.cassandra.locator.SimpleSeedProvider
    parameters:
         - seeds:  "10.168.66.41,10.176.170.59"
listen_address:
endpoint_snitch: GossipingPropertyFileSnitch
```

Note: Include at least one node from *each* datacenter in the seeds list.

5. In the `cassandra-rackdc.properties` file, assign the datacenter and rack names you determined in the Prerequisites. For example:

   Nodes 0 to 2

   ```
   ## Indicate the rack and dc for this node
   dc=DC1
   rack=RAC1
   ```

   Nodes 3 to 5

   ```
   ## Indicate the rack and dc for this node
   dc=DC2
   rack=RAC1
   ```

6. The `GossipingPropertyFileSnitch` always loads `cassandra-topology.properties` when that file is present. Remove the file from each node on any new cluster or any cluster migrated from the `PropertyFileSnitch`.

7. After you have installed and configured Cassandra on all nodes, DataStax recommends starting the seed nodes one at a time, and then starting the rest of the nodes.

   Note: If the node has restarted because of automatic restart, you must first stop the node and clear the data directories, as described above.

   Package installations:

   ```
   $ sudo service cassandra start
   ```

   Tarball installations:

   ```
   $ cd install_location
   $ bin/cassandra
   ```

8. To check that the ring is up and running, run:

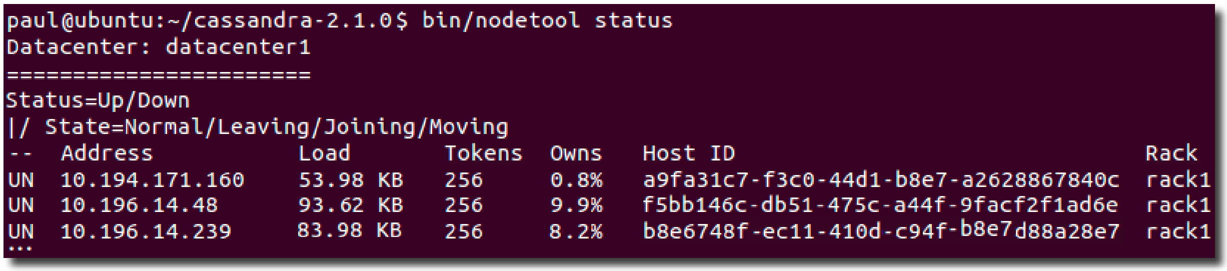
   Package installations:

   ```
   $ nodetool status
   ```

   Tarball installations:

   ```
   $ cd install_location
   $ bin/nodetool status
   ```

   The output should list each node, and show its status as `UN` (Up Normal).



```
paul@ubuntu:~/cassandra-2.1.0$ bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address          Load       Tokens  Owns   Host ID                               Rack
UN  10.194.171.160   53.98 KB   256     0.8%   a9fa31c7-f3c0-44d1-b8e7-a2628867840c  rack1
UN  10.196.14.48     93.62 KB   256     9.9%   f5bb146c-db51-475c-a44f-9facf2f1ad6e  rack1
UN  10.196.14.239    83.98 KB   256     8.2%   b8e6748f-ec11-410d-c94f-b8e7d88a28e7  rack1
...
```

Related information

Install locations on page 71

Starting and stopping Cassandra

Starting Cassandra as a service

Start the Cassandra Java server process for packaged installations.

Startup scripts are provided in the `/etc/init.d` directory. The service runs as the *cassandra* user.

Procedure

You must have root or sudo permissions to start Cassandra as a service.

On initial start-up, each node must be started one at a time, starting with your seed nodes:

```
$ sudo service cassandra start
```

Note: Cassandra 3.8 and later: Startup is aborted if corrupted transaction log files are found and the affected log files are logged. See the log files for information on resolving the situation.

On Enterprise Linux systems, the Cassandra service runs as a java process.

Starting Cassandra as a stand-alone process

Start the Cassandra Java server process for tarball installations.

Procedure

On initial start-up, each node must be started one at a time, starting with your seed nodes.

- To start Cassandra in the background:

  ```
  $ cd install_location
  $ bin/cassandra
  ```

  Note: Cassandra 3.8 and later: Startup is aborted if corrupted transaction log files are found and the affected log files are logged. See the log files for information on resolving the situation.
- To start Cassandra in the foreground:

  ```
  $ cd install_location
  $ bin/cassandra -f
  ```

Stopping Cassandra as a service

Stopping the Cassandra Java server process on packaged installations.

Procedure

You must have root or sudo permissions to stop the Cassandra service:

```
$ sudo service cassandra stop
```

Stopping Cassandra as a stand-alone process

Stop the Cassandra Java server process on tarball installations.

Procedure

Find the Cassandra Java process ID (PID), and then kill the process using its PID number:

```
$ ps auwx | grep cassandra
$ sudo kill pid
```

Clearing the data as a service

Remove all data from a package installation.

Procedure

To clear the data from the **default** directories:

After stopping the service, run the following command:

```
$ sudo rm -rf /var/lib/cassandra/*
```

Clearing the data as a stand-alone process

Remove data from a tarball installation.

Procedure

- To clear all data from the **default** directories, including the commitlog and saved_caches:

  a) Stop the process.

  b) Run the following command from the install directory:

  ```
  $ cd install_location
  $ sudo rm -rf data/*
  ```
- To clear the only the data directory:

  a) Stop the process.

  b) Run the following command from the install directory:

  ```
  $ cd install_location
  ```

```
$ sudo rm -rf data/data/*
```

Operations

Adding or removing nodes, datacenters, or clusters

Adding nodes to an existing cluster

Virtual nodes (vnodes) greatly simplify adding nodes to an existing cluster:

- Calculating tokens and assigning them to each node is no longer required.
- Rebalancing a cluster is no longer necessary because a node joining the cluster assumes responsibility for an even portion of the data.

For a detailed explanation about how vnodes work, see Virtual nodes on page 16.

Note: If you do not use vnodes, see Adding single-token nodes to a cluster on page 150.

Procedure

Be sure to use the same version of Cassandra on all nodes in the cluster. See Installing earlier releases.

1. Install Cassandra on the new nodes, but do not start Cassandra.

   If your Cassandra installation on Debian or Windows installation starts automatically, you must stop the node and clear the data.
2. Depending on the snitch used in the cluster, set either the properties in the cassandra-topology.properties or the cassandra-rackdc.properties file:

   - The PropertyFileSnitch uses the `cassandra-topology.properties` file.
   - The GossipingPropertyFileSnitch, Ec2Snitch, Ec2MultiRegionSnitch, and GoogleCloudSnitch use the `cassandra-rackdc.properties` file.
3. Set the following properties in the cassandra.yaml file:

auto_bootstrap

If this option has been set to false, you must set it to true. This option is not listed in the default `cassandra.yaml` configuration file and defaults to true.

cluster_name

The name of the cluster the new node is joining.

listen_address/broadcast_address

Can usually be left blank. Otherwise, use IP address or host name that other Cassandra nodes use to connect to the new node.

endpoint_snitch

The snitch Cassandra uses for locating nodes and routing requests.

num_tokens

The number of vnodes to assign to the node. If the hardware capabilities vary among the nodes in your cluster, you can assign a proportional number of vnodes to the larger machines.

seed_provider

Make sure that the new node lists at least one node in the existing cluster. The -seeds list determines which nodes the new node should contact to learn about the cluster and establish the gossip process.

Note: Seed nodes cannot bootstrap. Make sure the new node is not listed in the -seeds list. **Do not make all nodes seed nodes.** Please read Internode communications (gossip) on page 13.

Other non-default settings

Change any other non-default settings you have made to your existing cluster in the `cassandra.yaml` file and `cassandra-topology.properties` or `cassandra-rackdc.properties` files. Use the `diff` command to find and merge any differences between existing and new nodes.

4. Start the bootstrap node.
5. Use nodetool status to verify that the node is fully bootstrapped and all other nodes are up (UN) and not in any other state.
6. After all new nodes are running, run nodetool cleanup on each of the previously existing nodes to remove the keys that no longer belong to those nodes. Wait for cleanup to complete on one node before running nodetool cleanup on the next node.

   Cleanup can be safely postponed for low-usage hours.

Related tasks

Starting Cassandra as a service on page 138

Starting Cassandra as a stand-alone process on page 138

Related information

The nodetool utility on page 183

Install locations on page 71

Adding a datacenter to a cluster

Steps for adding a datacenter to an existing cluster.

Procedure

1. Configure the keyspace and create the new datacenter:

   a) Use ALTER KEYSPACE to configure all user-created, system, and DataStax Enterprise-defined keyspaces to use the NetworkTopologyStrategy.

   This is necessary for multiple datacenter clusters because nodetool rebuild (10 on page 142) requires a replica of these keyspaces in the specified source datacenter.

   Table: System and DataStax Enterprise-defined keyspaces

System keyspaces	DataStax Enterprise keyspaces
system_distributed	dse_perf
system_auth	dse_security
system_traces	dse_leases

   b) Create a new datacenter with a replication factor of `0`:

   You can use cqlsh to create or alter a keyspace:

```
CREATE KEYSPACE "sample-ks" WITH REPLICATION =
  { 'class' : 'NetworkTopologyStrategy', 'ExistingDC' : 3 }, 'NewDC1' :
 0;
```

2. Install Cassandra on each new node.

   Be sure to use the same version of Cassandra on all nodes in the cluster. See Installing earlier releases.
3. Configure cassandra.yaml on each new node:
   a) In the `cassandra.yaml` file, add the auto_bootstrap setting, and set it to `false`:

   ```
   auto_bootstrap: false
   ```

   Note: This property is not included in cassandra.yaml. By default, each node bootstraps when it starts, which means it gets data from other nodes in the datacenter. When starting a new datacenter, you do not want this to happen; instead, you want to add the appropriate data to each new node.

   b) Set other `cassandra.yaml` properties, such as -seeds and endpoint_snitch, to match the settings in the `cassandra.yaml` files on other nodes in the cluster. See Initializing a multiple node cluster (multiple datacenters) on page 135.

   Note: Do not make all nodes seeds, see Internode communications (gossip) on page 13.
4. On each new node, follow the configuration of the other nodes in the cluster to make the correct token assignment:
   a) To enable vnodes on the new node, set num_tokens. The recommended value is 256. Do not set the initial_token. DataStax Enterprise uses other values.
   b) To configure the node for single-node-per-token architecture, generate the initial token for each node. Add this as the value for each node's initial_token property, and make sure the num_tokens is commented out. See Generating tokens on page 130 and Adding or replacing single-token nodes.
5. On each new node, add the new datacenter definition to the properties file for the type of snitch used in the cluster:

Snitch	Configuring file
PropertyFileSnitch	`cassandra-topology.properties`
GossipPropertyFileSnitch	`cassandra-rackdc.properties`
Ec2Snitch	
Ec2MultiRegionSnitch	
GoogleCloudSnitch	

   You do not need to restart the node after adding the snitch configuration.
6. On each new node, remove the `cassandra-topology.properties` file as necessary.

   The `GossipingPropertyFileSnitch` always loads `cassandra-topology.properties` when that file is present. Remove the file from each node on any new cluster or any cluster migrated from the `PropertyFileSnitch`.
7. To ensure that your clients recognize the new datacenter, make sure they are configured to use the `DCAwareRoundRobinPolicy`. Review the programming instructions for your driver.
8. Review the consistency level for global or per-operation level for multiple datacenter operation:
   a) If using a `QUORUM` consistency level for reads or writes, check whether `LOCAL_QUORUM` or `EACH_QUORUM` meets your requirements for multiple datacenters.
   b) If using the `ONE` consistency level for reads or writes, check whether `LOCAL_ONE` consistency level meets your requirements for multiple datacenters.
9. Start Cassandra on the new nodes.
10. After all nodes are running in the cluster, run nodetool rebuild on each node in the new datacenter.

This step ensures that the new nodes recognize the existing datacenters in the cluster.

You can run rebuild on one or more nodes at the same time. The choice depends on whether your cluster can handle the extra I/O and network pressure of running on multiple nodes. Run on one node at a time to reduce the impact on the existing cluster.

```
$ nodetool rebuild -- name_of_existing_data_center
```

Attention: If you don't specify the existing datacenter in the command line, the new nodes will appear to rebuild successfully, but will not contain any data.

If you miss this step, requests to the new datacenter with LOCAL_ONE or ONE consistency levels may fail if the existing datacenters are not completely in-sync.

11. On each new node, remove the auto_bootstrap: false property in the `cassandra.yaml` file, or change its value to `true`.

    This returns this parameter to its normal setting, which allows the node to get all the data from the other nodes in the datacenter if it is restarted. You do not have to restart the node after changing this setting. It will take effect at the next restart.

Related tasks

Starting Cassandra as a service on page 138

Starting Cassandra as a stand-alone process on page 138

Related information

Install locations on page 71

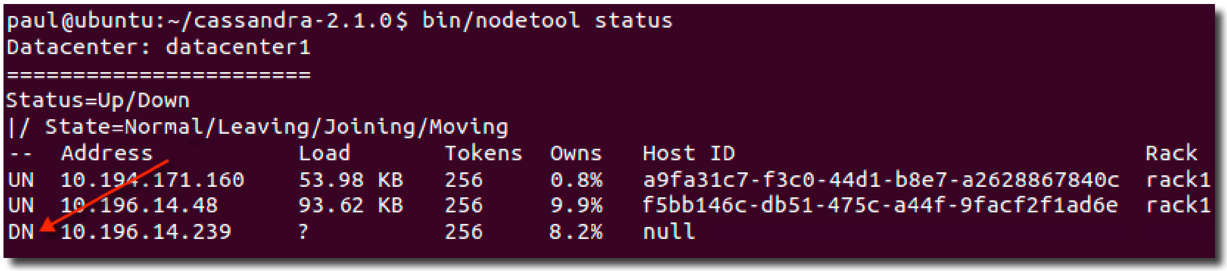
Replacing a dead node or dead seed node

Steps to replace a node that has died for some reason, such as hardware failure.

The procedure for replacing a dead node is the same for vnodes and single-token nodes. Extra steps are required for replacing dead seed nodes.

Procedure

1. Run nodetool status to verify that the node is dead (`DN`).



```
paul@ubuntu:~/cassandra-2.1.0$ bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address          Load       Tokens  Owns    Host ID                               Rack
UN  10.194.171.160   53.98 KB   256     0.8%    a9fa31c7-f3c0-44d1-b8e7-a2628867840c  rack1
UN  10.196.14.48     93.62 KB   256     9.9%    f5bb146c-db51-475c-a44f-9facf2f1ad6e  rack1
DN  10.196.14.239    ?          256     8.2%    null
```

2. Record the datacenter, address, and rack settings of the dead node; you will use these later.
3. Add the replacement node to the network and record its IP address.
4. If the dead node was a seed node, change the cluster's seed node configuration on each node:
   a) In the cassandra.yaml file for each node, remove the IP address of the dead node from the `- seeds` list in the seed-provider property.
   b) If the cluster needs a new seed node to replace the dead node, add the new node's IP address to the `- seeds` list.

Attention: In multiple data-center clusters, include at least one node from each datacenter (replication group) in the seed list. Designating more than a single seed node per datacenter is recommended for fault tolerance. Otherwise, gossip has to communicate with another datacenter when bootstrapping a node.

Making every node a seed node is **not** recommended because of increased maintenance and reduced gossip performance. Gossip optimization is not critical, but it is recommended to use a small seed list (approximately three nodes per datacenter).

c) Restart the node.

5. On an existing node, gather setting information for the new node from the cassandra.yaml file:
   - `cluster_name`
   - `endpoint_snitch`
   - Other non-default settings: Use the diff tool to compare current settings with default settings.
6. Gather rack and datacenter information:
   - If the cluster uses the PropertyFileSnitch, record the rack and data assignments listed in the cassandra-topology.properties file, or copy the file to the new node.
   - If the cluster uses the GossipPropertyFileSnitch, Ec2Snitch, Ec2MultiRegionSnitch, or GoogleCloudSnitch, record the rack and datacenter assignments in the dead node's `cassandra-rackdc.properties` file.
7. Make sure that the new node meets all prerequisites and then Install Cassandra on the new node, but do not start Cassandra.

   Note: Be sure to install the same version of Cassandra as is installed on the other nodes in the cluster. If not using the latest version, see Installing earlier releases.
8. If Cassandra automatically started on the node, stop and clear the data that was added automatically on startup.
9. Add values to the following properties in cassandara.yaml file from the information you gathered earlier:
   - auto_bootstrap: If this setting exists and is set to `false`, set it to `true`. (This setting is not included in the default `cassandra.yaml` configuration file.)
   - cluster_name
   - seed list
10. Add the rack and datacenter configuration:
    - If the cluster uses the GossipPropertyFileSnitch, Ec2Snitch, Ec2MultiRegionSnitch, or GoogleCloudSnitch:
      1. Add the dead node's rack and datacenter assignments to the cassandra-rackdc.properties file on the replacement node.

         Note: Do not remove the entry for the dead node's IP address yet.
      2. Delete the `cassandra-topology.properties` file.
    - If the cluster uses the PropertyFileSnitch:
      1. Copy the `cassandra-topology.properties` file from an existing node, or add the settings to the local copy.
      2. Edit the file to add an entry with the new node's IP address and the dead node's rack and datacenter assignments.
11. Start the new node with the replace_address option, passing in the IP address of the dead node.

    Package installations:

    1. Add the following option to cassandra-env.sh file:

       ```
       JVM_OPTS="$JVM_OPTS -Dcassandra.replace_address=address_of_dead_node
       ```
    2. Start the node.

3. After the node bootstraps, remove the `replace-address` parameter from `cassandra-env.sh`.
4. Restart the node.

Tarball installations:

- Start Cassandra with this option:

```
$ sudo bin/cassandra -Dcassandra.replace_address=address_of_dead_node
```

12. Run `nodetool status` to verify that the new node has bootstrapped successfully.
13. Wait at least 72 hours and then remove the old node's IP address from the `cassandra-topology.properties` or `cassandra-rackdc.properties` file.

    This ensures that old node's information is removed from gossip. If removed from the property file too soon, problems may result. Use nodetool gossipinfo to check the gossip status. The node is still in gossip until LEFT status disappears.

Replacing a running node

Steps to replace a node with a new node, such as when updating to newer hardware or performing proactive maintenance.

You can replace a running node in two ways:

- Adding a node and then decommissioning the old node on page 145
- Using nodetool to replace a running node on page 146

Note: To change the IP address of a node, simply change the IP of node and then restart Cassandra. If you change the IP address of a seed node, you must update the -seeds parameter in the seed_provider list in each node's cassandra.yaml file.

Adding a node and then decommissioning the old node

You must prepare and start the replacement node, integrate it into the cluster, and then decommission the old node.

Procedure

Be sure to use the same version of Cassandra on all nodes in the cluster. See Installing earlier releases.

1. Prepare and start the replacement node, as described in Adding nodes to an existing cluster.

   Note: If not using vnodes, see Adding single-token nodes to a cluster on page 150.
2. Confirm that the replacement node is alive:

   - Run nodetool ring if not using vnodes.
   - Run nodetool status if using vnodes.

   The status should show:

   - nodetool ring: `Up`
   - nodetool status: `UN`
3. Note the `Host ID` of the original node; it is used in the next step.
4. Using the Host ID of the original node, decommission the original node from the cluster using the nodetool decommission command.
5. Run nodetool cleanup on all the other nodes in the same datacenter.

Using nodetool to replace a running node

This method allows you to replace a running node while avoiding streaming the data twice or running cleanup.

CAUTION: If using a consistency level of ONE, you risk losing data because the node might contain the only copy of a record. Be absolutely sure that no application uses consistency level ONE.

Procedure

1. Stop Cassandra on the node to be replaced.
2. Follow the instructions for replacing a dead node using the old node’s IP address for `-Dcassandra.replace_address`.
3. Ensure that consistency level ONE is not used on this node.

Related tasks

Removing a node on page 147

Moving a node from one rack to another

A common task is moving a node from one rack to another. For example, when using GossipPropertyFileSnitch, a common error is mistakenly placing a node in the wrong rack. To correct the error, use one of the following procedures.

- The preferred method is to decommission the node and re-add it to the correct rack and datacenter.
  - This method takes longer to complete than the alternative method. Data is moved that the decommissioned node doesn't need anymore. Then the node gets new data while bootstrapping. The alternative method does both operations simultaneously.
- An alternative method is to update the node's topology and restart the node. Once the node is up, run a full repair on the cluster.

  CAUTION: This method is not preferred because until the repair is completed, the node may blindly handle requests for data the node doesn't yet have. To mitigate this problem with request handling, start the node with `-Dcassandra.join_ring=false` after repairing once, then fully join the node to the cluster using the JMX method `org.apache.cassandra.db.StorageService.joinRing()`. The node will be less likely to be out of sync with other nodes before it serves any requests. After joining the node to the cluster, repair the node again, so that any writes missed during the first repair will be captured.

Decommissioning a datacenter

Steps to properly remove a datacenter so no information is lost.

Procedure

1. Make sure no clients are still writing to any nodes in the datacenter.
2. Run a full repair with nodetool repair.

   This ensures that all data is propagated from the datacenter being decommissioned.
3. Change all keyspaces so they no longer reference the datacenter being removed.
4. Run nodetool decommission on every node in the datacenter being removed.

Removing a node

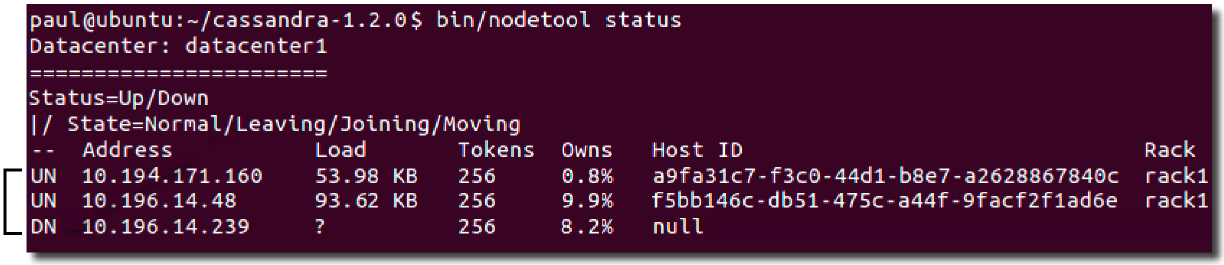
Use these instructions when you want to remove nodes to reduce the size of your cluster, not for replacing a dead node.

Attention: If you are not using virtual nodes (vnodes), you must rebalance the cluster.

Procedure

- Check whether the node is up or down using nodetool status:

  The nodetool command shows the status of the node (UN=up, DN=down):



```
paul@ubuntu:~/cassandra-1.2.0$ bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address          Load       Tokens  Owns   Host ID                               Rack
UN  10.194.171.160   53.98 KB   256     0.8%   a9fa31c7-f3c0-44d1-b8e7-a2628867840c  rack1
UN  10.196.14.48     93.62 KB   256     9.9%   f5bb146c-db51-475c-a44f-9facf2f1ad6e  rack1
DN  10.196.14.239    ?          256     8.2%   null
```

- If the node is up, run nodetool decommission.

  This assigns the ranges that the node was responsible for to other nodes and replicates the data appropriately.

  Use nodetool netstats to monitor the progress.
- If the node is down, choose the appropriate option:
  - If the cluster uses vnodes, remove the node using the nodetool removenode command.
  - If the cluster does not use vnodes, before running the nodetool removenode command, adjust your tokens to evenly distribute the data across the remaining nodes to avoid creating a hot spot.
- If `removenode` fails, run nodetool assassinate.

Switching snitches

Because snitches determine how Cassandra distributes replicas, the procedure to switch snitches depends on whether or not the topology of the cluster will change:

- If data has not been inserted into the cluster, there is no change in the network topology. This means that you only need to set the snitch; no other steps are necessary.
- If data has been inserted into the cluster, it's possible that the topology has changed and you will need to perform additional steps.
- If data has been inserted into the cluster that must be kept, change the snitch without changing the topology. Then add a new datacenter with new nodes and racks as desired. Finally, remove nodes from the old datacenters and racks. Simply altering the snitch and replication to move some nodes to a new datacenter will result in data being replicated incorrectly.

A change in topology means that there is a change in the datacenters and/or racks where the nodes are placed. Topology changes may occur when the replicas are placed in different places by the new snitch. Specifically, the replication strategy places the replicas based on the information provided by the new snitch. The following examples demonstrate the differences:

- **No topology change**

  Change from: five nodes using the SimpleSnitch in a single datacenter

To: five nodes in one datacenter and 1 rack using a network snitch such as the GossipingPropertyFileSnitch

- **Topology changes**
  - **Change from**: 5 nodes using the SimpleSnitch in a single datacenter

    To: 5 nodes in 2 datacenters using the PropertyFileSnitch (add a datacenter).

    Note: If "splitting" one datacenter into two, create a new datacenter with new nodes. Alter the keyspace replication settings for the keyspace that originally existed to reflect that two datacenters now exist. Once data is replicated to the new datacenter, remove the number of nodes from the original datacenter that have "moved" to the new datacenter.
  - **Change From**: 5 nodes using the SimpleSnitch in a single datacenter

    To: 5 nodes in 1 datacenter and 2 racks using the RackInferringSnitch (add rack information).

Procedure

Steps for switching snitches:

1. Create a properties file with datacenter and rack information.
   - `cassandra-rackdc.properties`

     GossipingPropertyFileSnitch on page 22, Ec2Snitch, and Ec2MultiRegionSnitch only.
   - `cassandra-topology.properties`

     All other network snitches.
2. Copy the cassandra-rackdc.properties or cassandra-topology.properties file to the Cassandra configuration directory on all the cluster's nodes. They won't be used until the new snitch is enabled.
3. Change the snitch for each node in the cluster in the node's cassandra.yaml file. For example:

   ```
   endpoint_snitch: GossipingPropertyFileSnitch
   ```
4. If the topology has not changed, you can restart each node one at a time.

   Any change in the `cassandra.yaml` file requires a node restart.
5. If the topology of the network has changed, but no datacenters are added:

   a) Shut down all the nodes, then restart them.

   b) Run a sequential repair and nodetool cleanup on each node.
6. If the topology of the network has changed and a datacenter is added:

   a) Create a new datacenter.

   b) Replicate data into new datacenter. Remove nodes from old datacenter.

   c) Run a sequential repair and nodetool cleanup on each node.

Related concepts

Snitches on page 20

Changing keyspace replication strategy

A keyspace is created with a strategy. For development work, the `SimpleStrategy` class is acceptable. For production work, the `NetworkTopologyStrategy` class must be set. To change the strategy, two steps are required. Altering the distribution of nodes within multiple datacenters when data is present should be accomplished by adding a datacenter, and then adding data to the new nodes in the new datacenter and removing nodes from the old datacenter.

Procedure

- Change the snitch to a network-aware setting.
- Alter the keyspace properties using the `ALTER KEYSPACE` command. For example, the keyspace cycling set to `SimpleStrategy` is switched to `NetworkTopologyStrategy` for a single datacenter `DC1`.

  ```
  cqlsh> ALTER KEYSPACE cycling WITH REPLICATION = {'class' :
   'NetworkTopologyStrategy', 'DC1' : 3};
  ```
- Alter the keyspace properties using the `ALTER KEYSPACE` command. For example, the keyspace cycling set to `SimpleStrategy` is switched to `NetworkTopologyStrategy`. Altering a keyspace to add a datacenter involves additional steps. Simply altering the keyspace may lead to faulty data replication. See switching snitches for additional information.

  ```
  cqlsh> ALTER KEYSPACE cycling WITH REPLICATION = {'class' :
   'NetworkTopologyStrategy', 'DC1' : 3, 'DC2' : 2 };
  ```
- Run nodetool-repair on each node that is affected by the change. For details, see Updating the replication factor

  It is possible to restrict the replication of a keyspace to selected datacenters, or a single datacenter. To do this, use the NetworkTopologyStrategy and set the replication factors of the excluded datacenters to `0` (zero), as in the following example:

  ```
  cqlsh> ALTER KEYSPACE cycling WITH REPLICATION = {'class' :
   'NetworkTopologyStrategy', 'DC1' : 0, 'DC2' : 3, 'DC3' : 0 };
  ```

Edge cases for transitioning or migrating a cluster

The information in this topic is intended for the following types of scenarios (without any interruption of service):

- Transition a cluster on EC2 to a cluster on Amazon virtual private cloud (VPC).
- Migrate from a cluster when the network separates the current cluster from the future location.
- Migrate from an early Cassandra cluster to a recent major version.

Procedure

The following method ensures that if something goes wrong with the new cluster, you still have the existing cluster until you no longer need it.

1. Set up and configure the new cluster as described in Initializing a cluster on page 132.

   Note: If you're not using vnodes, be sure to configure the token ranges in the new nodes to match the ranges in the old cluster.
2. Set up the schema for the new cluster using CQL.
3. Configure your client to write to both clusters.

   Depending on how the writes are done, code changes may be needed. Be sure to use identical consistency levels.
4. Ensure that the data is flowing to the new nodes so you won't have any gaps when you copy the snapshots to the new cluster in step 6.
5. Snapshot the old EC2 cluster.
6. Copy the data files from your keyspaces to the nodes.

- You may be able to copy the data files to their matching nodes in the new cluster, which is simpler and more efficient. This will work if:
    - You are not using vnodes
    - The destination is not a different version of Cassandra
    - The node ratio is 1:1
- If the clusters are different sizes or if you are using vnodes, use the sstableloader (Cassandra bulk loader) on page 273 (sstableloader).

7. You can either switch to the new cluster all at once or perform an incremental migration.

   For example, to perform an incremental migration, you can set your client to designate a percentage of the reads that go to the new cluster. This allows you to test the new cluster before decommissioning the old cluster.
8. Decommission the old cluster, as described in Decommissioning a datacenter on page 146.

Adding single-token nodes to a cluster

Steps for adding nodes in single-token architecture clusters, not vnodes.

To add capacity to a cluster, introduce new nodes in stages or by adding an entire datacenter. Use one of the following methods:

- **Add capacity by doubling the cluster size:** Adding capacity by doubling (or tripling or quadrupling) the number of nodes is less complicated when assigning tokens. Using this method, existing nodes keep their existing token assignments, and the new nodes are assigned tokens that bisect (or trisect) the existing token ranges.
- **Add capacity for a non-uniform number of nodes:** When increasing capacity with this method, you must recalculate tokens for the entire cluster, and assign the new tokens to the existing nodes.

Note: For DataStax Enterprise clusters, you can use OpsCenter to rebalance a cluster.

Procedure

1. Calculate the tokens for the nodes based on your expansion strategy using the Token Generating Tool.
2. Install Cassandra and configure Cassandra on each new node.
3. If Cassandra starts automatically (Debian), stop the node and clear the data.
4. Configure cassandra.yaml on each new node:
    - auto_bootstrap: If false, set it to true.

      This option is not listed in the default `cassandra.yaml` configuration file and defaults to true.
    - cluster_name
    - cassandra.yaml configuration file/broadcast_address: Usually leave blank. Otherwise, use the IP address or host name that other Cassandra nodes use to connect to the new node.
    - endpoint_snitch
    - initial_token: Set according to your token calculations.

      CAUTION: If this property has no value, Cassandra assigns the node a random token range and results in a badly unbalanced ring.
    - seed_provider: Make sure that the new node lists at least one seed node in the existing cluster.

      Seed nodes cannot bootstrap. Make sure the new nodes are not listed in the -seeds list. **Do not make all nodes seed nodes.** See Internode communications (gossip) on page 13.
    - Change any other non-default settings in the new nodes to match the existing nodes. Use the `diff` command to find and merge any differences between the nodes.

5. Depending on the snitch, assign the datacenter and rack names in the cassandra-topology.properties or cassandra-rackdc.properties for each node.
6. Start Cassandra on each new node in two minutes intervals with consistent.rangemovement turned off:

   - **Package installations:** To each bootstrapped node, add the following option to the `/usr/share/cassandra/cassandra-env.sh` file and then start Cassandra:

     ```
     JVM_OPTS="$JVM_OPTS -Dcassandra.consistent.rangemovement=false
     ```

   - **Tarball installations:**

     ```
     $ bin/cassandra -Dcassandra.consistent.rangemovement=false
     ```

The following operations are resource intensive and should be done during low-usage times.

7. After the new nodes are fully bootstrapped, use nodetool move to assign the new initial_token value to each node that requires one, one node at a time.
8. After all nodes have their new tokens assigned, run nodetool cleanup on each node in the cluster and wait for cleanup to complete on each node before doing the next node.

   This step removes the keys that no longer belong to the previously existing nodes.

Adding a datacenter to a single-token architecture cluster

Steps for adding a datacenter to single-token architecture clusters, not vnodes.

Procedure

1. Ensure that you are using NetworkTopologyStrategy for all keyspaces.
2. For each new node, edit the configuration properties in the cassandra.yaml file:

   - Set `auto_bootstrap` to `False`.
   - Set the `initial_token`. Be sure to offset the tokens in the new datacenter, see Generating tokens on page 130.
   - Set the `cluster name`.
   - Set any other non-default settings.
   - Set the seed lists. Every node in the cluster must have the same list of seeds and include at least one node from each datacenter. Typically one to three seeds are used per datacenter.
3. Update either the properties file on all nodes to include the new nodes. You do not need to restart.

   - GossipingPropertyFileSnitch: cassandra-rackdc.properties
   - PropertyFileSnitch: cassandra-topology.properties
4. Ensure that your client does not auto-detect the new nodes so that they aren't contacted by the client until explicitly directed.
5. If using a QUORUM consistency level for reads or writes, check the LOCAL_QUORUM or EACH_QUORUM consistency level to make sure that the level meets the requirements for multiple datacenters.
6. Start the new nodes.
7. The `GossipingPropertyFileSnitch` always loads `cassandra-topology.properties` when that file is present. Remove the file from each node on any new cluster or any cluster migrated from the `PropertyFileSnitch`.
8. After all nodes are running in the cluster:

   a) Change the replication factor for your keyspace for the expanded cluster.
   b) Run nodetool rebuild on each node in the new datacenter.

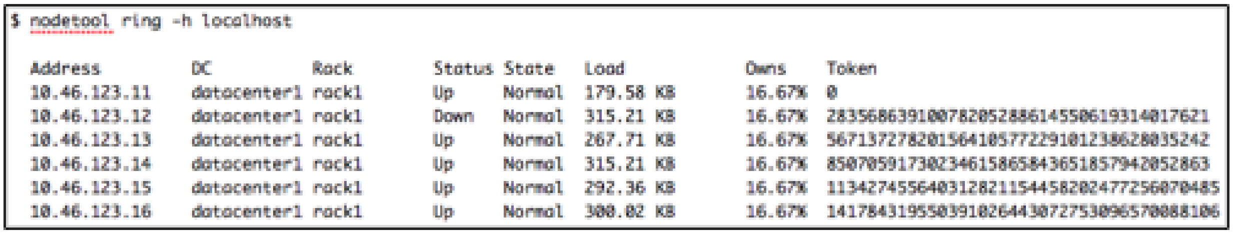
Replacing a dead node in a single-architecture cluster

Steps for replacing nodes in single-token architecture clusters, not vnodes.

Procedure

1. Confirm that the node is dead using nodetool ring on any live node in the cluster.

   A *Down* status indicates the dead node:

   ```
   $ nodetool ring -h localhost

     Address         DC          Rack        Status State   Load            Owns    Token
     10.46.123.11    datacenter1 rack1       Up     Normal  179.58 KB       16.67%  0
     10.46.123.12    datacenter1 rack1       Down   Normal  315.21 KB       16.67%  28356863910078205288614550619314017621
     10.46.123.13    datacenter1 rack1       Up     Normal  267.71 KB       16.67%  56713727820156410577229101238628035242
     10.46.123.14    datacenter1 rack1       Up     Normal  315.21 KB       16.67%  85070591730234615865843651857942052863
     10.46.123.15    datacenter1 rack1       Up     Normal  292.36 KB       16.67%  113427455640312821154458202477256070485
     10.46.123.16    datacenter1 rack1       Up     Normal  300.02 KB       16.67%  141784319550391026443072753096570088106
   ```
2. Install Cassandra on the replacement node.
3. Remove any pre-existing Cassandra data on the replacement node:

   ```
   $ sudo rm -rf /var/lib/cassandra/*
   ```
4. Set `auto_bootstrap: true`.

   If `auto_bootstrap` is not in the cassandra.yaml file, it automatically defaults to `true`.
5. Set the `initial_token` in the `cassandra.yaml` file to the value of the dead node's token -1.

   ```
   initial_token: 28356863910078205288614550619314017620
   ```
6. Configure any non-default settings in the node's `cassandra.yaml` to match your existing cluster.
7. Start the new node.
8. After the new node has finished bootstrapping, check that it is up using `nodetool ring`.
9. Run nodetool repair on each keyspace to ensure the node is fully consistent:

   ```
   $ nodetool repair -h 10.46.123.12 keyspace_name
   ```
10. Remove the dead node.

Backing up and restoring data

About snapshots

Cassandra backs up data by taking a snapshot of all on-disk data files (SSTable files) stored in the data directory. You can take a snapshot of all keyspaces, a single keyspace, or a single table while the system is online.

Using a parallel ssh tool (such as pssh), you can snapshot an entire cluster. This provides an *eventually consistent* backup. Although no one node is guaranteed to be consistent with its replica nodes at the time a snapshot is taken, a restored snapshot resumes consistency using Cassandra's built-in consistency mechanisms.

After a system-wide snapshot is performed, you can enable incremental backups on each node to backup data that has changed since the last snapshot: each time a memtable is flushed to disk and an SSTable is created, a hard link is copied into a `/backups` subdirectory of the data directory (provided JNA is enabled). Compacted SSTables will not create hard links in `/backups` because these SSTables do not contain any data that has not already been linked.

Taking a snapshot

Snapshots are taken per node using the nodetool snapshot command. To take a global snapshot, run the nodetool snapshot command using a parallel ssh utility, such as pssh.

A snapshot first flushes all in-memory writes to disk, then makes a hard link of the SSTable files for each keyspace. You must have enough free disk space on the node to accommodate making snapshots of your data files. A single snapshot requires little disk space. However, snapshots can cause your disk usage to grow more quickly over time because a snapshot prevents old obsolete data files from being deleted. After the snapshot is complete, you can move the backup files to another location if needed, or you can leave them in place.

Note: Cassandra can only restore data from a snapshot when the table schema exists. It is recommended that you also backup the schema. See `DESCRIBE SCHEMA` in DESCRIBE.

Procedure

Run the nodetool snapshot command, specifying the hostname, JMX port, and keyspace. For example:

```
$ nodetool -h localhost -p 7199 snapshot mykeyspace
```

Results

The snapshot is created in `data_directory/`*`keyspace_name`*`/`*`table_name`*`-`*`UUID`*`/snapshots/`*`snapshot_name`* directory. Each snapshot directory contains numerous `.db` files that contain the data at the time of the snapshot.

For example:

- Package installations: `/var/lib/cassandra/data/mykeyspace/users-081a1500136111e482d09318a3b15cc2/snapshots/1406227071618/mykeyspace-users-ka-1-Data.db`
- Tarball installations: *`install_location`*`/data/data/mykeyspace/users-081a1500136111e482d09318a3b15cc2/snapshots/1406227071618/mykeyspace-users-ka-1-Data.db`

Deleting snapshot files

When taking a snapshot, previous snapshot files are not automatically deleted. You should remove old snapshots that are no longer needed.

The nodetool clearsnapshot command removes all existing snapshot files from the snapshot directory of each keyspace. You should make it part of your back-up process to clear old snapshots before taking a new one.

Procedure

1. To delete all snapshots for a node, run the nodetool `clearsnapshot` command. For example:

```
$ nodetool -h localhost -p 7199 clearsnapshot
```

To delete snapshots on all nodes at once, run the nodetool clearsnapshot command using a parallel ssh utility.

2. To delete a single snapshot, run the `clearsnapshot` command with the snapshot name:

   ```
   $ nodetool clearsnapshot -t <snapshot_name>
   ```

   The file name and path vary according to the type of snapshot. See nodetools snapshot for details about snapshot names and paths.

Enabling incremental backups

When incremental backups are enabled (disabled by default), Cassandra hard-links each memtable-flushed SSTable to a backups directory under the keyspace data directory. This allows storing backups offsite without transferring entire snapshots. Also, incremental backups combined with snapshots to provide a dependable, up-to-date backup mechanism. Compacted SSTables will not create hard links in `/backups` because these SSTables do not contain any data that has not already been linked.A snapshot at a point in time, plus all incremental backups and commit logs since that time form a compete backup.

As with snapshots, Cassandra does not automatically clear incremental backup files. DataStax recommends setting up a process to clear incremental backup hard-links each time a new snapshot is created.

Procedure

Edit the cassandra.yaml configuration file on each node in the cluster and change the value of incremental_backups to true.

Restoring from a snapshot

Restoring a keyspace from a snapshot requires all snapshot files for the table, and if using incremental backups, any incremental backup files created after the snapshot was taken. Streamed SSTables (from repair, decommission, and so on) are also hardlinked and included.

Note: Restoring from snapshots and incremental backups temporarily causes intensive CPU and I/O activity on the node being restored.

Restoring from local nodes

This method copies the SSTables from the snapshots directory into the correct data directories.

1. Make sure the table schema exists.

   Cassandra can only restore data from a snapshot when the table schema exists. If the schema does not exist and has not been backed up, you must recreate the schema.
2. If necessary, truncate the table.

   Note: You may not need to truncate under certain conditions. For example, if a node lost a disk, you might restart before restoring so that the node continues to receive new writes before starting the restore procedure.

   Truncating is usually necessary. For example, if there was an accidental deletion of data, the tombstone from that delete has a later write timestamp than the data in the snapshot. If you restore without truncating (removing the tombstone), Cassandra continues to shadow the restored data. This behavior also occurs for other types of overwrites and causes the same problem.
3. Locate the most recent snapshot folder. For example:

`data_directory/keyspace_name/table_name-UUID/snapshots/snapshot_name`

4. Copy the most recent snapshot SSTable directory to the `data_directory/keyspace/table_name-UUID directory`.
5. Run nodetool refresh.

Restoring from centralized backups

This method uses sstableloader to restore snapshots.

1. Make sure the table schema exists.

   Cassandra can only restore data from a snapshot when the table schema exists. If the schema does not exist and has not been backed up, you must recreate the schema.
2. If necessary, truncate the table.

   Note: You may not need to truncate under certain conditions. For example, if a node lost a disk, you might restart before restoring so that the node continues to receive new writes before starting the restore procedure.

   Truncating is usually necessary. For example, if there was an accidental deletion of data, the tombstone from that delete has a later write timestamp than the data in the snapshot. If you restore without truncating (removing the tombstone), Cassandra continues to shadow the restored data. This behavior also occurs for other types of overwrites and causes the same problem.
3. Restore the most recent snapshot using the sstableloader tool on the backed-up SSTables.

   The sstableloader streams the SSTables to the correct nodes. You do not need to remove the commitlogs or drain or restart the nodes.

Restoring a snapshot into a new cluster

Suppose you want to copy a snapshot of SSTable data files from a three node Cassandra cluster with vnodes enabled (256 tokens) and recover it on another newly created three node cluster (256 tokens). The token ranges will not match, because the token ranges cannot be exactly the same in the new cluster. You need to specify the tokens for the new cluster that were used in the old cluster.

Note: This procedure assumes you are familiar with restoring a snapshot and configuring and initializing a cluster.

Procedure

To recover the snapshot on the new cluster:

1. From the old cluster, retrieve the list of tokens associated with each node's IP:

   ```
   $ nodetool ring | grep ip_address_of_node | awk '{print $NF ","}' | xargs
   ```
2. In the cassandra.yaml file for each node in the new cluster, add the list of tokens you obtained in the previous step to the initial_token parameter using the same num_tokens setting as in the old cluster.
3. Make any other necessary changes in the new cluster's cassandra.yaml and property files so that the new nodes match the old cluster settings. Make sure the seed nodes are set for the new cluster.
4. Clear the system table data from each new node:

   ```
   $ sudo rm -rf /var/lib/cassandra/data/system/*
   ```

   This allows the new nodes to use the initial tokens defined in the cassandra.yaml when they restart.
5. Start each node using the specified list of token ranges in new cluster's cassandra.yaml:

   ```
   initial_token: -9211270970129494930, -9138351317258731895,
    -8980763462514965928, ...
   ```

6. Create schema in the new cluster. All the schemas from the old cluster must be reproduced in the new cluster.
7. Stop the node. Using `nodetool refresh` is unsafe because files within the data directory of a running node can be silently overwritten by identically named just-flushed SSTables from memtable flushes or compaction. Copying files into the data directory and restarting the node will not work for the same reason.
8. Restore the SSTable files snapshotted from the old cluster onto the new cluster using the same directories, while noting that the UUID component of target directory names has changed. Without restoration, the new cluster will not have data to read upon restart.
9. Restart the node.

Recovering from a single disk failure using JBOD

Steps for recovering from a single disk failure in a disk array using JBOD (just a bunch of disks).

Cassandra might not fail from the loss of one disk in a JBOD array, but some reads and writes may fail when:

- The operation's consistency level is ALL.
- The data being requested or written is stored on the defective disk.
- The data to be compacted is on the defective disk.

It's possible that you can simply replace the disk, restart Cassandra, and run `nodetool repair`. However, if the disk crash corrupted the Cassandra system table, you must remove the incomplete data from the other disks in the array. The procedure for doing this depends on whether the cluster uses vnodes or single-token architecture.

Procedure

These steps are supported for Cassandra versions 3.2 and later. If a disk fails on a node in a cluster using an earlier version of Cassandra, replace the node.

1. Verify that the node has a defective disk and identify the disk:
   a) Check the logs on the affected node.

      Disk failures are logged in `FILE NOT FOUND` entries, which identifies the mount point or disk that has failed.

   b) If no `FILE NOT FOUND` entries exist, see the DSE Troubleshooting.
2. If the node is still running, stop Cassandra and shut down the node.
3. Replace the defective disk and restart the node.
4. If the node cannot restart:
   a) Try restarting Cassandra without bootstrapping the node:

      Package installations:

      1. Add the following option to cassandra-env.sh file:

         ```
         JVM_OPTS="$JVM_OPTS -Dcassandra.allow_unsafe_replace=true
         ```

      2. Start the node.
      3. After the node bootstraps, remove the `-Dcassandra.allow_unsafe_replace=true` parameter from `cassandra-env.sh`.
      4. Restart the node.

      Tarball installations:

      - Start Cassandra with this option:

```
$ sudo bin/cassandra Dcassandra.allow_unsafe_replace=true
```

5. If Cassandra restarts, run nodetool repair on the node. If not, replace the node.
6. If the repair succeeds, the node is restored to production. Otherwise, go to 7 on page 157 or 8 on page 157.
7. For a cluster using vnodes:
   a) On the affected node, clear the `system` directory on each functioning drive.

      Example for a node with a three disk JBOD array:

      ```
      $ -/mnt1/cassandra/data
      $ -/mnt2/cassandra/data
      $ -/mnt3/cassandra/data
      ```

      If `mnt1` has failed:

      ```
      $ rm -fr /mnt2/cassandra/data/system
      $ rm -fr /mnt3/cassandra/data/system
      ```

   b) Restart Cassandra without bootstrapping as described in 4 on page 156:

      ```
      $ -Dcassandra.allow_unsafe_replace=true
      ```

   c) Run nodetool repair on the node.

      If the repair succeeds, the node is restored to production. If not, replace the dead node.
8. For a cluster single-token nodes:
   a) On one of the cluster's working nodes, run nodetool ring to retrieve the list of the repaired node's tokens:

      ```
      $ nodetool ring | grep ip_address_of_node | awk ' {print $NF ","}' | xargs
      ```

   b) Copy the output of the `nodetool ring` into a spreadsheet (space-delimited).

   c) Edit the output, keeping the list of tokens and deleting the other columns.

   d) On the node with the new disk, open the cassandra.yaml file and add the tokens (as a comma-separated list) to the initial_token property.

   e) Change any other non-default settings in the new nodes to match the existing nodes. Use the `diff` command to find and merge any differences between the nodes.

      If the repair succeeds, the node is restored to production. If not, replace the node.

   f) On the affected node, clear the `system` directory on each functioning drive.

      Example for a node with a three disk JBOD array:

      ```
      $ -/mnt1/cassandra/data
      $ -/mnt2/cassandra/data
      $ -/mnt3/cassandra/data
      ```

      If `mnt1` has failed:

      ```
      $ rm -fr /mnt2/cassandra/data/system
      $ rm -fr /mnt3/cassandra/data/system
      ```

   g) Restart Cassandra without bootstrapping as described in 4 on page 156:

      ```
      $ -Dcassandra.allow_unsafe_replace=true
      ```

   h) Run nodetool repair on the node.

      If the repair succeeds, the node is restored to production. If not, replace the node.

Repairing nodes

Over time, data in a replica can become inconsistent with other replicas due to the distributed nature of the database. Node repair corrects the inconsistencies so that eventually all nodes have the same and most up-to-date data. It is important part of regular maintenance for every Cassandra cluster.

Cassandra provides the following repair processes:

- Hinted Handoff

  If a node becomes unable to receive a particular write, the write's coordinator node preserves the data to be written as a set of *hints*. When the node comes back online, the coordinator effects repair by handing off hints so that the node can catch up with the required writes.
- Read Repair

  During the read path, a query assembles data from several nodes. The coordinator node for this write compares the data from each replica node. If any replica node has outdated data, the coordinator node sends it the most recent version. The scope of this type of repair depends on the keyspace's replication factor. During a write, Cassandra collects only enough replica data to satisfy the replication factor, and only performs read repair on nodes that participate in that write operation.

  But Cassandra can also choose a write at random and perform read repair on all replicas, regardless of the replication factor.
- Anti-Entropy Repair

  Cassandra provides the nodetool repair tool, which you can use to repair recovering nodes, and which you should use as part of regular maintenance purposes.

You can use Cassandra settings or Cassandra tools to configure each type of repair. For details on when to use each type of repair and how to configure each one, see the pages listed above.

Hinted Handoff: repair during write path

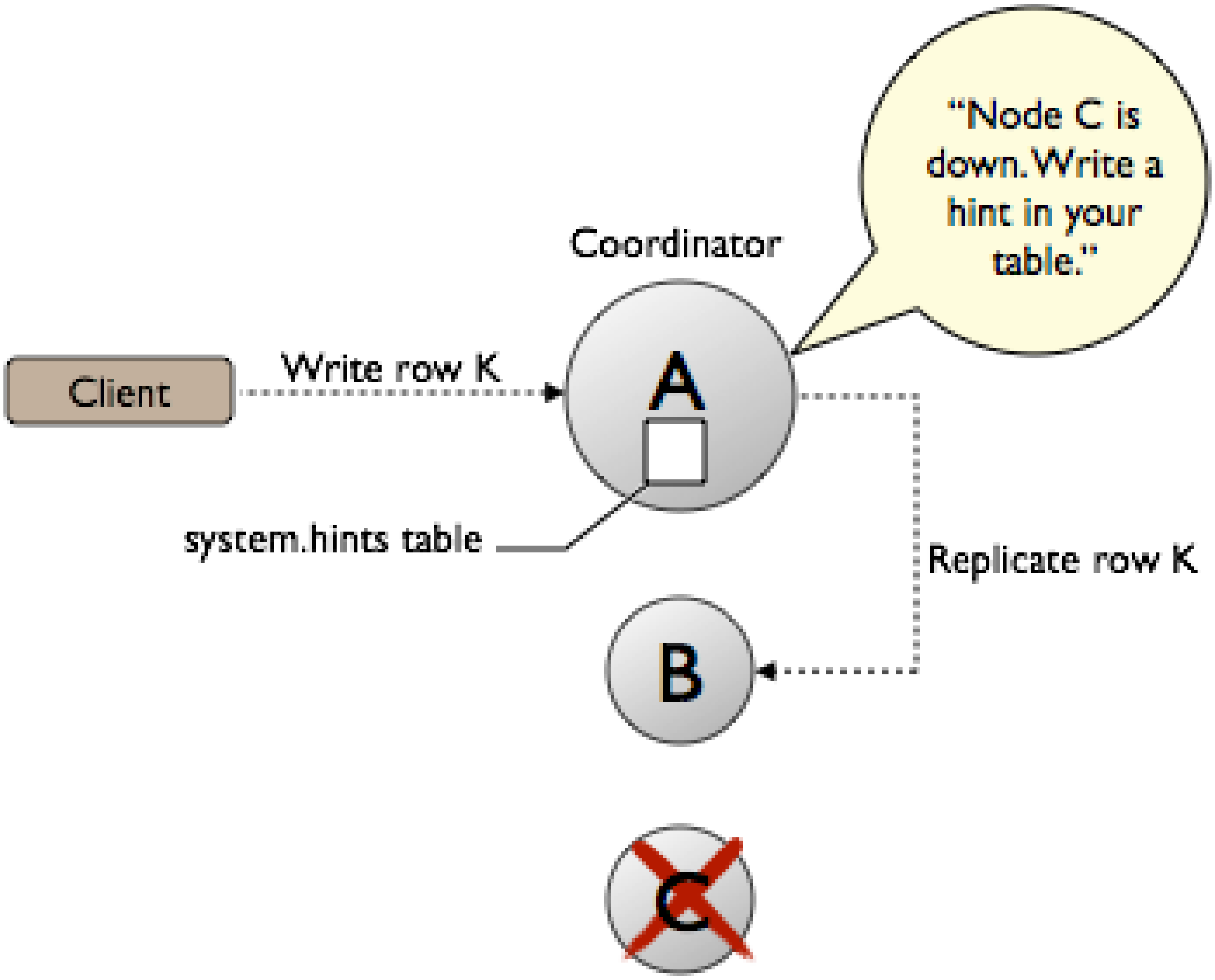
On occasion, a node becomes unresponsive while data is being written. Reasons for unresponsiveness are hardware problems, network issues, or overloaded nodes that experience long garbage collection (GC) pauses. By design, hinted handoff inherently allows Cassandra to continue performing the same number of writes even when the cluster is operating at reduced capacity.

After the failure detector marks a node as down, missed writes are stored by the coordinator for a period of time, if hinted handoff is enabled in the cassandra.yaml file. In Cassandra 3.0 and later, the hint is stored in a local hints directory on each node for improved replay. The hint consists of a target ID for the downed node, a hint ID that is a time UUID for the data, a message ID that identifies the Cassandra version, and the data itself as a blob. Hints are flushed to disk every 10 seconds, reducing the staleness of the hints. When gossip discovers when a node has comes back online, the coordinator replays each remaining hint to write the data to the newly-returned node, then deletes the hint file. If a node is down for longer than max_hint_window_in_ms (3 hours by default), the coordinator stops writing new hints.

The coordinator also checks every ten minutes for hints corresponding to writes that timed out during an outage too brief for the failure detector to notice through gossip. If a replica node is overloaded or unavailable, and the failure detector has not yet marked the node as down, then expect most or all writes to that node to fail after the timeout triggered by write_request_timeout_in_ms, (10 seconds by default). The coordinator returns a `TimeOutException` exception, and the write will fail but a hint will be stored. If several nodes experience brief outages simultaneously, substantial memory pressure can build up on the coordinator. The coordinator tracks how many hints it is currently writing, and if the number increases too much, the coordinator refuses writes and throws the `OverloadedException` exception.

The consistency level of a write request affects whether hints are written and a write request subsequently fails. If the cluster consists of two nodes, A and B, with a replication factor of 1, each row is stored on only one node. Suppose node A is the coordinator, but goes down before a row K is written to it with a consistency level of ONE. In this case, the consistency level specified cannot be met, and since node A is the coordinator, it cannot store a hint. Node B cannot write the data, because it has not received the data as the coordinator nor has a hint been stored. The coordinator checks the number of replicas that are up and will not attempt to write the hint if the consistency level specified by a client cannot be met. A hinted handoff failure occurs and will return a `UnavailableException` exception. The write request fails and the hint is not written.

In general, the recommendation is to have enough nodes in the cluster and a replication factor sufficient to avoid write request failures. For example, consider a cluster consisting of three nodes, A, B, and C,with a replication factor of 2. When a row K is written to the coordinator (node A in this case), even if node C is down, the consistency level of ONE or QUORUM can be met. Why? Both nodes A and B will receive the data, so the consistency level requirement is met. A hint is stored for node C and written when node C comes up. In the meantime, the coordinator can acknowledge that the write succeeded.



For applications that want Cassandra to accept writes when all the normal replicas are down and consistency level ONE cannot be satisfied, Cassandra provides consistency level ANY. ANY guarantees that the write is durable and readable after an appropriate replica target becomes available and receives the hint replay.

Nodes that die might have stored undelivered hints, because any node can be a coordinator. The data on the dead node will be stale after a long outage as well. If a node has been down for an extended period of time, a manual repair should be run.

At first glance, it seems that hinted handoff eliminates the need for manual repair, but this is not true because hardware failure is inevitable and has the following ramifications:

- Loss of the historical data necessary to tell the rest of the cluster exactly what data is missing.
- Loss of hints-not-yet-replayed from requests that the failed node coordinated.

When removing a node from the cluster by decommissioning the node or by using the nodetool removenode command, Cassandra automatically removes hints targeting the node that no longer exists. Cassandra also removes hints for dropped tables.

For more explanation about hint storage, see What's Coming to Cassandra in 3.0: Improved Hint Storage and Delivery or an older blog that discusses the basics, Modern hinted handoff.

Read Repair: repair during read path

Read repair improves consistency in a Cassandra cluster with every read request.

In a read, the coordinator node sends a data request to one replica node and digest requests to others for consistency level (CL) greater than 1). If all nodes return consistent data, the coordinator returns it to the client. For a description of how Cassandra handles inconsistency among replicas, see How are read requests accomplished? on page 48.

In read repair, Cassandra sends a digest request to each replica not directly involved in the read. Cassandra compares all replicas and writes the most recent version to any replica node that does not have it. If the query's consistency level is above `ONE`, Cassandra performs this process on all replica nodes in the foreground before the data is returned to the client. Read repair fixes anything it touches. This means that for CL ONE, nothing is fixed because no comparison takes place. For QUORUM, only the nodes that the query touches are repaired, not ALL.

Cassandra can also perform read repair randomly on a table, independent of any read. You can configure how often this happens using the read_repair_chance property for a table.

Read repair cannot be performed on tables that use DateTieredCompactionStrategy, because of the way timestamps are checked for DTCS compaction. If your table uses DateTieredCompactionStrategy, set `read_repair_chance` to zero. For other compaction strategies, `read_repair_chance` is typically set to a value of 0.2.

Manual repair: Anti-entropy repair

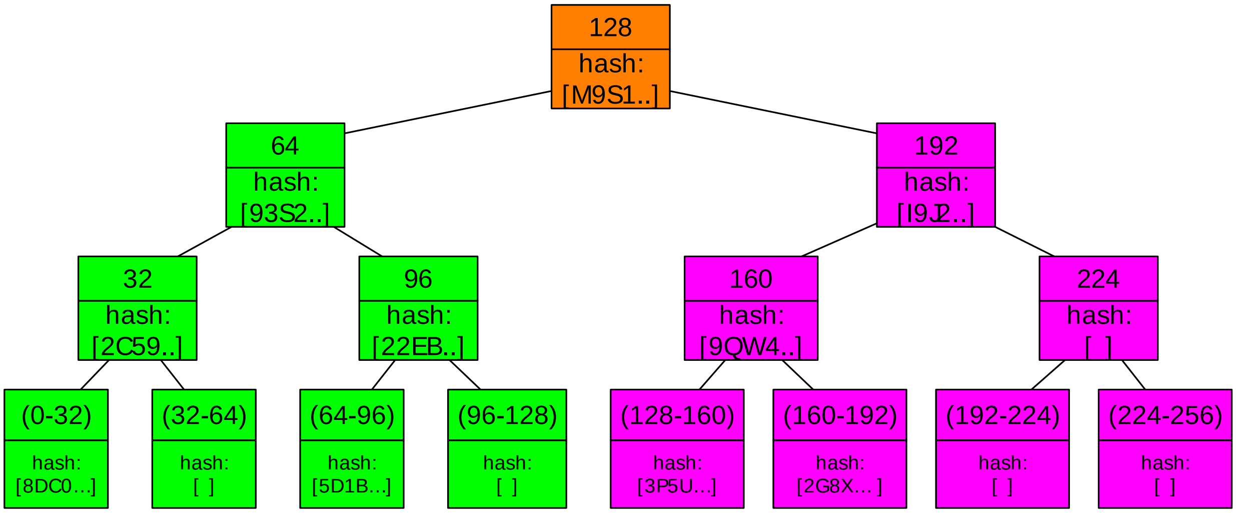
Anti-entropy node repairs are important for every Cassandra cluster. Frequent data deletions and downed nodes are common causes of data inconsistency. Use anti-entropy repair for routine maintenance and when a cluster needs fixing by running the nodetool repair command.

How does anti-entropy repair work?

Cassandra accomplishes anti-entropy repair using Merkle trees, similar to Dynamo and Riak. Anti-entropy is a process of comparing the data of all replicas and updating each replica to the newest version. Cassandra has two phases to the process:

1. Build a Merkle tree for each replica
2. Compare the Merkle trees to discover differences

Merkle trees are binary hash trees whose leaves are hashes of the individual key values. The leaf of a Cassandra Merkle tree is the hash of a row value. Each Parent node higher in the tree is a hash of its respective children. Because higher nodes in the Merkle tree represent data further down the tree, Casandra can check each branch independently without requiring the coordinator node to download the entire data set. For anti-entropy repair Cassandra uses a compact tree version with a depth of 15 (2^15 = 32K leaf nodes). For example, a node containing a million partitions with one damaged partition, about 30 partitions are streamed, which is the number that fall into each of the *leaves* of the tree. Cassandra works with smaller Merkle trees because they require less storage memory and can be transferred more quickly to other nodes during the comparison process.



Row key: jack	Row key: jill	Row key: terry	Row key: misty
Row token: 5	Row token: 7	Row token: 10	Row token: 20
hash: 8DC0....	hash: 5D1B...	hash: 3P5U...	hash: @G8X...

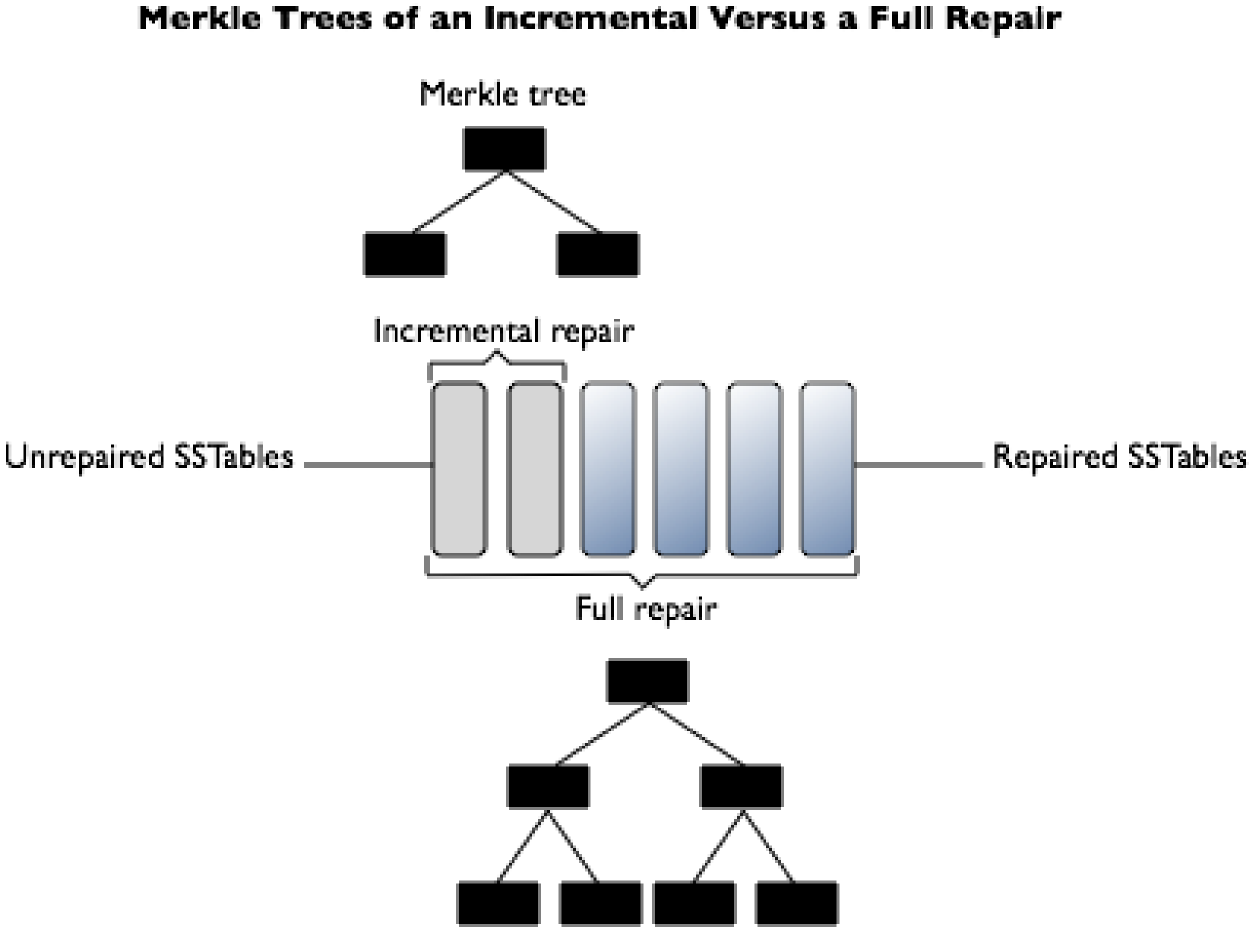
After the initiating node receives the Merkle trees from the participating peer nodes, the initiating node compares every tree to every other tree. If a difference is detected, the differing nodes exchange data for the conflicting range(s), and the new data is written to SSTables. The comparison begins with the top node of the Merkle tree. If no difference is detected, the process proceeds to the left child node and compares and then the right child node. When a node is found to differ, inconsistent data exists for the range that pertains to that node. All data that corresponds to the leaves below that Merkle tree node will be replaced with new data. For any given replica set, Cassandra performs validation compaction on only one replica at a time.

Merkle tree building is quite resource intensive, stressing disk I/O and using memory. Some of the options discussed here help lessen the impact on the cluster performance.

The `nodetool repair` command can be run on either a specified node or on all nodes if a node is not specified. The node that initiates the repair becomes the coordinator node for the operation. To build the Merkle trees, the coordinator node determines peer nodes with matching ranges of data. A major, or validation, compaction is triggered on the peer nodes. The validation compaction reads and generates a hash for every row in the stored column families, adds the result to a Merkle tree, and returns the tree to the initiating node. Merkle trees use hashes of the data, because in general, hashes will be smaller than the data itself. Repair in Cassandra discusses this process in more detail.

Full vs Incremental repair

Th section above describes a full repair of a node's data: Cassandra compares all SSTables for that node and makes necessary repairs. The default setting is incremental repair. An incremental repair persists data that has already been repaired, and only builds Merkle trees for unrepaired SSTables. This more efficient process depends on new metadata that marks the rows in an SSTable as repaired or unrepaired.



Reducing the size of the Merkle tree improves the performance of the incremental repair process, assuming repairs are run frequently. Incremental repairs work like full repairs, with an initiating node requesting Merkle trees from peer nodes with the same unrepaired data, and then comparing the Merkle trees to discover mismatches. Once the data has been reconciled and new SSTables built, the initiating node issues an anti-compaction command. Anti-compaction is the process of segregating repaired and unrepaired ranges into separate SSTables, unless the SSTable fits entirely within the repaired range. In the latter case, the SSTable metadata `repairedAt` is updated to reflect its repaired status.

Anti-compaction is handled differently, depending on the compaction strategy assigned to the data.

- Size-tiered compaction (STCS) splits repaired and unrepaired data into separate pools for separate compactions. A major compaction generates two SSTables, one for each pool of data.
- Leveled compaction (LCS) performs size-tiered compaction on unrepaired data. After repair completes, Casandra moves data from the set of unrepaired SSTables to L0.
- Date-tiered (DTCS) splits repaired and unrepaired data into separate pools for separate compactions. A major compaction generates two SSTables, one for each pool of data. DTCS compaction should not use incremental repair.

Full repair is the default in Cassandra 2.1 and earlier. Incremental repair is the default for Cassandra 2.2 and later. In Cassandra 2.2 and later, when a full repair is run, SSTables are marked as repaired and anti-compacted.

Parallel vs Sequential repair

Sequential repair takes action on one node after another. Parallel repair repairs all nodes with the same replica data at the same time.

Sequential repair takes a snapshot of each replica. Snapshots are hardlinks to existing SSTables. They are immutable and require almost no disk space. The snapshots are active while the repair proceeds, then Cassandra deletes them. When the coordinator node finds discrepancies in the Merkle trees, the coordinator node makes required repairs from the snapshots. For example, for a table in a keyspace with a Replication factor RF=3 and replicas A, B and C, the `repair` command takes a snapshot of each replica immediately and then repairs each replica from the snapshots sequentially (using snapshot A to repair replica B, then snapshot A to repair replica C, then snapshot B to repair replica C).

Parallel repair works on nodes A, B, and C all at once. During parallel repair, the dynamic snitch processes queries for this table using a replica in the snapshot that is not undergoing repair.

Sequential repair is the default in Cassandra 2.1 and earlier. Parallel repair is the default for Cassandra 2.2 and later.

Note: Sequential and incremental do not work together in Cassandra 2.1.

Partitioner range (`-pr`)

Within a cluster, Cassandra stores a particular range of data on multiple nodes. If you run `nodetool repair` on one node at a time, Cassandra may repair the same range of data several times (depending on the replication factor used in the keyspace). If you use the partitioner range option (`-pr`), `nodetool repair` only repairs a specified range of data once, rather than repeating the repair operation. This decreases the strain on network resources, although `nodetool repair` still builds Merkle trees for each replica.

Note: If you use this option, you must run `nodetool repair -pr` on every node in the cluster to repair all data. Otherwise, some ranges of data will not be repaired.

The partitioner range option is recommended for routine maintenance. Do not use it to repair a downed node. Do not use with incremental repair (default for Cassandra 3.0 and later).

Local (`-local, --in-local-dc`) vs datacenter (`dc, --in-dc`) vs Cluster-wide repair

Consider carefully before using `nodetool repair` across datacenters, instead of within a local datacenter. When you run repair locally on a node using `-local` or `--in-local-dc`, the command runs only on nodes within the same datacenter as the node that runs it. Otherwise, the command runs cluster-wide repair processes on all nodes that contain replicas, even those in different datacenters. For example, if you start `nodetool repair` over two datacenters, DC1 and DC2, each with a replication factor of 3, `repair`must build Merkle tables for 6 nodes. The number of Merkle Tree increases linearly for additional datacenters. Cluster-wide repair also increases network traffic between datacenters tremendously, and can cause cluster issues.

If the local option is too limited, consider using the `-dc` or `--in-dc` options, limiting repairs to a specific datacenter. This does not repair replicas on nodes in other datacenters, but it can decrease network traffic while repairing more nodes than the local options.

The `nodetool repair -pr` option is good for repairs across multiple datacenters.

Additional notes for `-local` repairs:

- The `nodetool repair` tool does not support the use of `-local` with the `-pr` option unless the datacenter's nodes have all the data for all ranges.
- Also, the tool does not support the use of `-local` with `-inc` (incremental repair).

Note: For Cassandra 2.2 and later, a recommended option for repairs across datacenters: use the `-dcpar` or `--dc-parallel` to repair datacenters in parallel.

Endpoint range vs Subrange repair (`-st, --start-token, -et --end-token`)

A repair operation runs on all partition ranges on a node, or endpoint range, unless you use the `-st` and `-et` (or `-start-token` and `-end-token`) options to run subrange repairs. When you specify a start token and end token, `nodetool repair` works between these tokens, repairing only those partition ranges.

Subrange repair is not a good strategy because it requires generated token ranges. However, if you know which partition has an error, you can target that partition range precisely for repair. This approach can relieve the problem known as overstreaming, which ties up resources by sending repairs to a range over and over.

Subrange repair involves more than just the `nodetool repair` command. A Java `describe_splits` call to ask for a split containing 32k partitions can be iterated throughout the entire range incrementally or in parallel to eliminate the overstreaming behavior. Once the tokens are generated for the split, they are passed to `nodetool repair -st <start_token> -et <end_token>`. The `-local` option can be used to repair only within a local data center to reduce cross data center transfer.

When to run anti-entropy repair

When to run anti-entropy repair is dependent on the characteristics of a Cassandra cluster. General guidelines are presented here, and should be tailored to each particular case.

When is repair needed?

Run repair in these situations:

- To routinely maintain node health.

  Note: Even if deletions never occur, schedule regular repairs. Setting a column to null is a delete.
- To recover a node after a failure while bringing it back into the cluster.
- To update data on a node containing data that is not read frequently, and therefore does not get read repair.
- To update data on a node that has been down.
- To recover missing data or corrupted SSTables. To do this, you must run non-incremental repair.

Guidelines for running routine node repair include:

- Run incremental repair daily, run full repairs weekly to monthly. Monthly is generally sufficient, but run more frequently if warranted.

  Important: Full repair is useful for maintaining data integrity, even if deletions never occur.
- Use the parallel and partitioner range options, unless precluded by the scope of the repair.
- Run a full repair to eliminate anti-compaction. Anti-compaction is the process of splitting an SSTable into two SSTables, one with repaired data and one with non-repaired data. This has compaction strategy implications.

  Note: Before you can run incremental repair, you must set the repaired state of each SSTable. For instructions, see Migrating to incremental repairs.
- Run repair frequently enough that every node is repaired before reaching the time specified in the gc_grace_seconds setting. Deleted data is properly handled in the cluster if this requirement is met.
- Schedule routine node repair to minimize cluster disruption.
  - If possible, schedule repair operation for low-usage hours.
  - If possible, schedule repair operations on single nodes at a time.
- Increase the time value setting of gc_grace_seconds if data is seldom deleted or overwritten. For these tables, changing the setting will:
  - Minimizes impact to disk space.
  - Allow longer interval between repair operations.

- Mitigate heavy disk usage by configuring nodetool compaction throttling options (setcompactionthroughput and setcompactionthreshold) before running a repair.

Guidelines for running repair on a downed node:

- Do not use partitioner range, `-pr`.

Migrating to incremental repairs

Repairing SSTables using anti-entropy repairis a necessary part of Cassandra maintenance. A full repair of all SSTables on a node takes a lot of time and is resource-intensive. You can manage repairs with less service disruption using incremental repair. Incremental repair consumes less time and resources because it skips SSTables that are already marked as repaired.

Incremental repair works equally well with any compaction scheme — Size-Tiered Compaction (STCS), Date-Tiered Compaction(DTCS), Time-Window Compaction(TWCS), or Leveled Compaction (LCS).

In Cassandra 3.0 and later, switching from full repair to incremental repair is easier than before. However, the first system-wide incremental repair can take a long time, as Cassandra recompacts all SSTables according to the chosen compaction shceme. You can make this process less disruptive by *migrating* the cluster to incremental repair one node at a time.

Overview of the procedure

To migrate one Cassandra node to incremental repair:

1. Disable autocompaction on the node.
2. Run a full, sequential repair.
3. Stop the node.
4. Set the `repairedAt` metadata value to each SSTable that existed before you disabled compaction.
5. Restart Cassandra on the node.
6. Re-enable autocompaction on the node.

Prerequisites

Listing SSTables

Before you run a full repair on the node, list its SSTables. The existing SSTables may not be changed by the repair process, and the incremental repair process you run later will not process these SSTables unless you set the `repairedAt` value for each SSTable (see Step 4 below).

You can find the node's SSTables in one of the following locations:

This directory contains a subdirectory for each keyspace. Each of these subdirectories contains a set of files for each SSTable. The name of the file that contains the SSTable data has the following format:

```
<version_code>-<generation>-<format>-Data.db
```

Note: You can mark multiple SSTables as a batch by running `sstablerepairedset` with a text file of filenames — see Step 4.

Migrating the node to incremental repair

Note: In RHEL and Debianß installations, you must install the tools packages before you can follow these steps.

1. **Disable autocompaction on the node**

From the *install_directory*:

```
$ bin/nodetool disableautocompaction
```

Running this command without parameters disables autocompaction for all keyspaces. For details, see disableautocompaction.

2. **Run the default full, sequential repair**

   From the *install_directory*:

   ```
   $ bin/nodetool repair
   ```

   Running this command without parameters starts a full sequential repair of all SSTables on the node. This may take a substantial amount of time. For details, see repair.
3. **Stop the node**.
4. **Set the `repairedAt` metadata value to each SSTable that existed before you disabled compaction**.

   Use sstablerepairedset on page 280. To mark a single SSTable *SSTable-example-Data.db*:

   ```
   sudo bin/sstablerepairedset --really-set --is-repaired SSTable-example-
   Data.db
   ```

   To do this as a batch process using a text file of SSTable names:

   ```
   sudo bin/sstablerepairedset --really-set --is-repaired -f SSTable-
   names.txt
   ```

   Note: The value of the `repairedAt` metadata is the timestamp of the last repair. The `sstablerepairedset` command applies the current date/time. To check the value of the `repairedAt` metadata for an SSTable, use:

   ```
   $ bin/sstablemetadata example-keyspace-SSTable-example-Data.db | grep
    "Repaired at"
   ```
5. **Restart the node**.

What to do next

After you have migrated all nodes, you will be able to run incremental repairs using `nodetool repair` with the -inc parameter. For details, see http://www.datastax.com/dev/blog/more-efficient-repairs.

Related information

http://www.datastax.com/dev/blog/repair-in-cassandra

http://www.datastax.com/dev/blog/more-efficient-repairs

http://www.datastax.com/dev/blog/anticompaction-in-cassandra-2-1

Monitoring Cassandra

Monitoring a Cassandra cluster

Understanding the performance characteristics of a Cassandra cluster is critical to diagnosing issues and planning capacity.

Cassandra exposes a number of statistics and management operations via Java Management Extensions (JMX). JMX is a Java technology that supplies tools for managing and monitoring Java applications and services. Any statistic or operation that a Java application has exposed as an MBean can then be monitored or manipulated using JMX.

JMX). JMX is a Java technology that supplies tools for managing and monitoring Java applications and services. Any statistic or operation that a Java application has exposed as an MBean can then be monitored or manipulated using JMX.

During normal operation, Cassandra outputs information and statistics that you can monitor using JMX-compliant tools, such as:

- The Cassandra nodetool utility
- JConsole

Using the same tools, you can perform certain administrative commands and operations such as flushing caches or doing a node repair.

Monitoring using the nodetool utility

The nodetool utility is a command-line interface for monitoring Cassandra and performing routine database operations. It is typically run from an operational Cassandra node.

The nodetool utility supports the most important JMX metrics and operations, and includes other useful commands for Cassandra administration, such as the proxyhistogram command. This example shows the output from nodetool proxyhistograms after running 4,500 insert statements and 45,000 select statements on a three ccm node-cluster on a local computer.

```
$ nodetool proxyhistograms

proxy histograms
Percentile      Read Latency     Write Latency     Range Latency
                    (micros)          (micros)          (micros)
50%                  1502.50            375.00            446.00
75%                  1714.75            420.00            498.00
95%                 31210.25            507.00            800.20
98%                 36365.00            577.36            948.40
99%                 36365.00            740.60           1024.39
Min                   616.00            230.00            311.00
Max                 36365.00          55726.00          59247.00
```

For a summary of the ring and its current state of general health, use the status command. For example:

```
$ nodetool status

Note: Ownership information does not include topology; for complete
 information, specify a keyspace
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Tokens  Owns    Host ID
    Rack
UN  127.0.0.1  47.66 KB   1       33.3%   aaa1b7c1-6049-4a08-
ad3e-3697a0e30e10  rack1
UN  127.0.0.2  47.67 KB   1       33.3%   1848c369-4306-4874-
afdf-5c1e95b8732e  rack1
UN  127.0.0.3  47.67 KB   1       33.3%   49578bf1-728f-438d-b1c1-
d8dd644b6f7f  rack1
```

The nodetool utility provides commands for viewing detailed metrics for tables, server metrics, and compaction statistics:

- `nodetool tablestats` displays statistics for each table and keyspace.
- `nodetool tablehistograms` provides statistics about a table, including read/write latency, row size, column count, and number of SSTables.
- `nodetool netstats` provides statistics about network operations and connections.
- `nodetool tpstats` provides statistics about the number of active, pending, and completed tasks for each stage of Cassandra operations by thread pool.

Monitoring using JConsole

JConsole is a JMX-compliant tool for monitoring Java applications such as Cassandra. It is included with Sun JDK 5.0 and higher. JConsole consumes the JMX metrics and operations exposed by Cassandra and displays them in a well-organized GUI. For each node monitored, JConsole provides these six separate tab views:

- Overview

  Displays overview information about the Java VM and monitored values.
- Memory

  Displays information about memory use.
- Threads

  Displays information about thread use.
- Classes

  Displays information about class loading.
- VM Summary

  Displays information about the Java Virtual Machine (VM).
- Mbeans

  Displays information about MBeans.

The Overview and Memory tabs contain information that is very useful for Cassandra developers. The Memory tab allows you to compare heap and non-heap memory usage, and provides a control to immediately perform Java garbage collection.

For specific Cassandra metrics and operations, the most important area of JConsole is the MBeans tab. This tab lists the following Cassandra MBeans:

- org.apache.cassandra.auth

  Includes permissions cache.
- org.apache.cassandra.db

  Includes caching, table metrics, and compaction.
- org.apache.cassandra.internal

  Internal server operations such as gossip, hinted handoff, and Memtable values.
- org.apache.cassandra.metrics

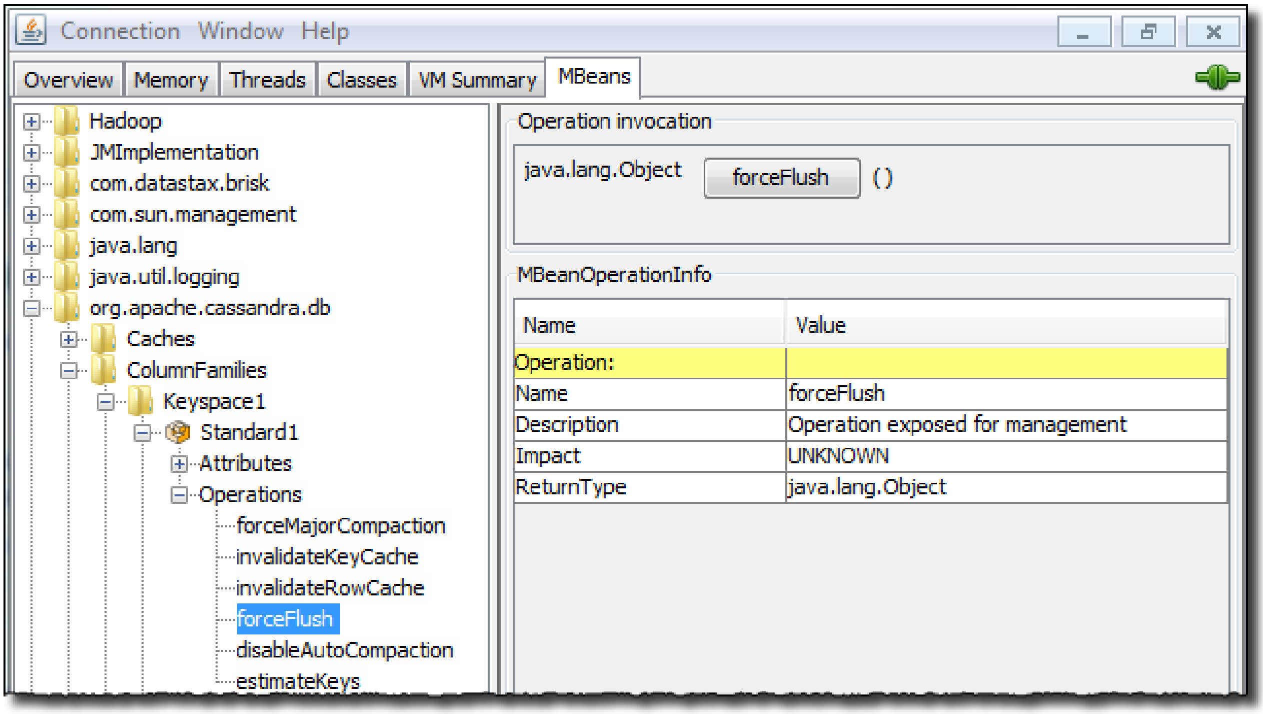
  Includes metrics on CQL, clients, keyspaces, read repair, storage, and threadpools and other topics.
- org.apache.cassandra.net

  Inter-node communication including FailureDetector, MessagingService and StreamingManager.
- org.apache.cassandra.request

  Tasks related to read, write, and replication operations.
- org.apache.cassandra.service

  Includes GCInspector.

When you select an MBean in the tree, its MBeanInfo and MBean Descriptor are displayed on the right, and any attributes, operations or notifications appear in the tree below it. For example, selecting and expanding the org.apache.cassandra.db MBean to view available actions for a table results in a display like the following:



Name	Value
Operation:	
Name	forceFlush
Description	Operation exposed for management
Impact	UNKNOWN
ReturnType	java.lang.Object

If you choose to monitor Cassandra using JConsole, keep in mind that JConsole consumes a significant amount of system resources. For this reason, DataStax recommends running JConsole on a remote machine rather than on the same host as a Cassandra node.

The JConsole CompactionManagerMBean exposes compaction metrics that can indicate when you need to add capacity to your cluster.

Compaction metrics

Monitoring compaction performance is an important aspect of knowing when to add capacity to your cluster. The following attributes are exposed through CompactionManagerMBean:

Table: Compaction Metrics

Attribute	Description
BytesCompacted	Total number of bytes compacted since server [re]start
CompletedTasks	Number of completed compactions since server [re]start
PendingTasks	Estimated number of compactions remaining to perform
TotalCompactionsCompleted	Total number of compactions since server [re]start

Thread pool and read/write latency statistics

Cassandra maintains distinct thread pools for different stages of execution. Each of the thread pools provide statistics on the number of tasks that are active, pending, and completed. Trends on these pools for increases in the pending tasks column indicate when to add additional capacity. After a baseline is established, configure alarms for any increases above normal in the pending tasks column. Use nodetool tpstats on the command line to view the thread pool details shown in the following table.

Table: Thread Pool statistics reported by nodetool tpstats

Thread Pool	Description
AntiEntropyStage	Tasks related to repair
CacheCleanupExecutor	Tasks related to cache maintenance (counter cache, row cache)
CompactionExecutor	Tasks related to compaction
CounterMutationStage	Tasks related to leading counter writes
GossipStage	Tasks related to the gossip protocol
HintsDispatcher	Tasks related to sending hints
InternalResponseStage	Tasks related to miscellaneous internal task responses
MemtableFlushWriter	Tasks related to flushing memtables
MemtablePostFlush	Tasks related to maintenance after memtable flush completion
MemtableReclaimMemory	Tasks related to reclaiming memtable memory
MigrationStage	Tasks related to schema maintenance
MiscStage	Tasks related to miscellaneous tasks, including snapshots and removing hosts
MutationStage	Tasks related to writes
Native-Transport-Requests	Tasks related to client requests from CQL
PendingRangeCalculator	Tasks related to recalculating range ownership after bootstraps/ decommissions
PerDiskMemtableFlushWriter_*	Tasks related to flushing memtables to a given disk
ReadRepairStage	Tasks related to performing read repairs
ReadStage	Tasks related to reads
RequestResponseStage	Tasks for callbacks from intra-node requests
Sampler	Tasks related to sampling statistics
SecondaryIndexManagement	Tasks related to secondary index maintenance
ValidationExecutor	Tasks related to validation compactions
ViewMutationStage	Tasks related to maintaining materialized views

Read/Write latency metrics

Cassandra tracks latency (averages and totals) of read, write, and slicing operations at the server level through StorageProxyMBean.

Table statistics

For individual tables, ColumnFamilyStoreMBean provides the same general latency attributes as StorageProxyMBean. Unlike StorageProxyMBean, ColumnFamilyStoreMBean has a number of other statistics that are important to monitor for performance trends. The most important of these are:

Table: Table Statistics

Attribute	Description
MemtableDataSize	The total size consumed by this table's data (not including metadata).
MemtableColumnsCount	Returns the total number of columns present in the memtable (across all keys).
MemtableSwitchCount	How many times the memtable has been flushed out.
RecentReadLatencyMicros	The average read latency since the last call to this bean.
RecentWriterLatencyMicros	The average write latency since the last call to this bean.
LiveSSTableCount	The number of live SSTables for this table.

The recent read latency and write latency counters are important in making sure operations are happening in a consistent manner. If these counters start to increase after a period of staying flat, you probably need to add capacity to the cluster.

You can set a threshold and monitor LiveSSTableCount to ensure that the number of SSTables for a given table does not become too great.

Tuning Java resources

Tuning the Java Virtual Machine (JVM) can improve performance or reduce high memory consumption.

Topics in this page:

- About garbage collection on page 171
- Choosing a Java garbage collector on page 171
- Setting G1 as the Java garbage collector on page 172
- Determining the heap size on page 172
- How Cassandra uses memory on page 173 - Read first for a better understanding of the settings and recommendations in this topic.
- Adjusting JVM parameters for other Cassandra services on page 174
- Other JMX options on page 174

About garbage collection

Garbage collection is the process by which Java removes data that is no longer needed from memory. To achieve the best performance, it is important to select the right garbage collector and heap size settings.

One situation that you definitely want to minimize is a garbage collection pause, also known as a stop-the-world event. A pause occurs when a region of memory is full and the JVM needs to make space to continue. During a pause all operations are suspended. Because a pause affects networking, the node can appear as down to other nodes in the cluster. Additionally, any Select and Insert statements will wait, which increases read and write latencies. Any pause of more than a second, or multiple pauses within a second that add to a large fraction of that second, should be avoided. The basic cause of the problem is the rate of data stored in memory outpaces the rate at which data can be removed. For specific symptoms and causes, see Garbage collection pauses.

Choosing a Java garbage collector

For Cassandra 3.0 and later, using the Concurrent-Mark-Sweep (CMS) or G1 garbage collector depends on these factors:

G1 is recommended in the following circumstances and reasons:

- Heap sizes from 14 GB to 64 GB.

  G1 performs better than CMS for larger heaps because it scans the regions of the heap containing the most garbage objects first, and compacts the heap on-the-go, while CMS stops the application when performing garbage collection.
- The workload is variable, that is, the cluster is performing the different processes all the time.
- For future proofing, as CMS will be deprecated in Java 9.
- G1 is easier to configure.
- G1 is self tuning.

CMS is recommended in the following circumstances:

- You have the time and expertise to manually tune and test garbage collection.

  Be aware that allocating more memory to the heap, can result in diminishing performance as the garbage collection facility increases the amount of Cassandra metadata in heap memory.
- Heap sizes no larger than 14 GB.
- The workload is fixed, that is, the cluster performs the same processes all the time.
- The environment requires the lowest latency possible. G1 incurs some latency due to profiling.

Note: For help configuring CMS, contact the DataStax Services team.

Setting G1 as the Java garbage collector

1. Open jvm.options.
2. Comment out the `-Xmn800M` line.
3. Comment out all lines in the `### CMS Settings` section.
4. Uncomment the relevant G1 settings in the `### G1 Settings` section:

```
## Use the Hotspot garbage-first collector.
-XX:+UseG1GC
#
## Have the JVM do less remembered set work during STW, instead
## preferring concurrent GC. Reduces p99.9 latency.
#-XX:G1RSetUpdatingPauseTimePercent=5
```

Note: When using G1, you only need to set MAX_HEAP_SIZE.

Determining the heap size

You might be tempted to set the Java heap to consume the majority of the computer's RAM. However, this can interfere with the operation of the OS page cache. Recent operating systems maintain the OS page cache for frequently accessed data and are very good at keeping this data in memory. Properly tuning the OS page cache usually results in better performance than increasing the Cassandra row cache.

Cassandra automatically calculates the maximum heap size (MAX_HEAP_SIZE) based on this formula:

```
max(min(1/2 ram, 1024MB), min(1/4 ram, 8GB)
```

For production use, you may wish to adjust heap size for your environment using the following guidelines:

- Heap size is usually between ¼ and ½ of system memory.
- Do not devote all memory to heap because it is also used for offheap cache and file system cache.
- Always enable GC logging when adjusting GC.
- Adjust settings gradually and test each incremental change.
- Enable parallel processing for GC, particularly when using DSE Search.
- Cassandra's `GCInspector` class logs information about any garbage collection that takes longer than 200 ms. Garbage collections that occur frequently and take a moderate length of time (seconds)

to complete, indicate excessive garbage collection pressure on the JVM. In addition to adjusting the garbage collection options, other remedies include adding nodes, and lowering cache sizes.
- For a node using G1, the Cassandra community recommends a MAX_HEAP_SIZE as large as possible, up to 64 GB.

Note: For more tuning tips, see Secret HotSpot option improving GC pauses on large heaps.

MAX_HEAP_SIZE

The recommended maximum heap size depends on which GC is used:

Hardware setup	Recommended MAX_HEAP_SIZE
Older computers	Typically 8 GB.
CMS for newer computers (8+ cores) with up to 256 GB RAM	No more 14 GB.
G1 for newer computers (8+ cores) with up to 256 GB RAM	14 GB to 64 GB.

The easiest way to determine the optimum heap size for your environment is:

1. Set the maximum heap size in the jvm.options file to a high arbitrary value on a single node. For example:

   ```
   -Xms48G
   -Xmx48G
   ```

   Set the min (-Xms) and max (-Xmx) heap sizes to the same value to avoid stop-the-world GC pauses during resize, and to lock the heap in memory on startup which prevents any of it from being swapped out.
2. Enable GC logging.
3. Check the logs to view the heap used by that node and use that value for setting the heap size in the cluster:

Note: This method decreases performance for the test node, but generally does not significantly reduce cluster performance.

If you don't see improved performance, contact the DataStax Services team for additional help.

HEAP_NEWSIZE

For CMS, you may also need to adjust HEAP_NEWSIZE. This setting determines the amount of heap memory allocated to newer objects or *young generation*. Cassandra calculates the default value for this property (in MB) as the lesser of:

- 100 times the number of cores
- ¼ of MAX_HEAP_SIZE

As a starting point, set HEAP_NEWSIZE to 100 MB per physical CPU core. For example, for a modern 8-core+ machine:

```
-Xmn800M
```

A larger HEAP_NEWSIZE leads to longer GC pause times. For a smaller HEAP_NEWSIZE, GC pauses are shorter but usually more expensive.

How Cassandra uses memory

Cassandra performs the following major operations within JVM heap:

- To perform reads, Cassandra maintains the following components in heap memory:

  - Bloom filters
  - Partition summary
  - Partition key cache
  - Compression offsets
  - SSTable index summary

  This metadata resides in memory and is proportional to total data. Some of the components grow proportionally to the size of total memory.
- Cassandra gathers replicas for a read or for anti-entropy repair and compares the replicas in heap memory.
- Data written to Cassandra is first stored in memtables in heap memory. Memtables are flushed to SSTables on disk.

To improve performance, Cassandra also uses off-heap memory as follows:

- Page cache. Cassandra uses additional memory as page cache when reading files on disk.
- The Bloom filter and compression offset maps reside off-heap.
- Cassandra can store cached rows in native memory, outside the Java heap. This reduces JVM heap requirements, which helps keep the heap size in the sweet spot for JVM garbage collection performance.

Adjusting JVM parameters for other Cassandra services

- **Solr**: Some Solr users have reported that increasing the stack size improves performance under Tomcat.

  To increase the stack size, uncomment and modify the default setting in the cassandra-env.sh file.

  ```
  # Per-thread stack size.
  JVM_OPTS="$JVM_OPTS -Xss256k"
  ```

  Also, decreasing the memtable space to make room for Solr caches can improve performance. Modify the memtable space by changing the memtable_heap_space_in_mb and memtable_offheap_space_in_mb properties in the cassandra.yaml file.
- **MapReduce**: Because MapReduce runs outside the JVM, changes to the JVM do not affect Analytics/Hadoop operations directly.

Other JMX options

Cassandra exposes other statistics and management operations via Java Management Extensions (JMX). JConsole and the nodetool utility are JMX-compliant management tools.

Configure Cassandra for JMX management by editing these properties in cassandra-env.sh.

- `com.sun.management.jmxremote.port`: sets the port on which Cassandra listens from JMX connections.
- `com.sun.management.jmxremote.ssl`: enables or disables SSL for JMX.
- `com.sun.management.jmxremote.authenticate`: enables or disables remote authentication for JMX.
- `-Djava.rmi.server.hostname`: sets the interface hostname or IP that JMX should use to connect. Uncomment and set if you are having trouble connecting.

Note: By default, you can interact with Cassandra using JMX on port 7199 without authentication.

Data caching

Configuring data caches

Cassandra includes integrated caching and distributes cache data around the cluster. When a node goes down, the client can read from another cached replica of the data. The integrated architecture also facilitates troubleshooting because there is no separate caching tier, and cached data matches what is in the database exactly. The integrated cache alleviates the cold start problem by saving the cache to disk periodically. Cassandra reads contents back into the cache and distributes the data when it restarts. The cluster does not start with a cold cache.

The saved key cache files include the ID of the table in the file name. A saved key cache filename for the `users` table in the `mykeyspace` keyspace looks similar to: `mykeyspace-users.users_name_idx-19bd7f80352c11e4aa6a57448213f97f-KeyCache-b.db2046071785672832311.tmp`

About the partition key cache

The partition key cache is a cache of the partition index for a Cassandra table. Using the key cache instead of relying on the OS page cache decreases seek times. Enabling just the key cache results in disk (or OS page cache) activity to actually read the requested data rows, but not enabling the key cache results in more reads from disk.

About the row cache

Note: Utilizing appropriate OS page cache will result in better performance than using row caching. Consult resources for page caching for the operating system on which Cassandra is hosted.

Configure the number of rows to cache in a partition by setting the rows_per_partition table option. To cache rows, if the row key is not already in the cache, Cassandra reads the first portion of the partition, and puts the data in the cache. If the newly cached data does not include all cells configured by user, Cassandra performs another read. The actual size of the row-cache depends on the workload. You should properly benchmark your application to get ”the best” row cache size to configure.

There are two row cache options, the old serializing cache provider and a new off-heap cache (OHC) provider. The new OHC provider has been benchmarked as performing about 15% better than the older option.

Using key cache and row cache

Typically, you enable either the partition key or row cache for a table.

Tip: Enable a row cache only when the number of reads is much bigger (rule of thumb is 95%) than the number of writes. Consider using the operating system page cache instead of the row cache, because writes to a partition invalidate the whole partition in the cache.

Tip: Disable caching entirely for archive tables, which are infrequently read.

Enabling and configuring caching

Use CQL to enable or disable caching by configuring the caching table property. Set parameters in the cassandra.yaml file to configure global caching properties:

- Partition key cache size
- Row cache size
- How often Cassandra saves partition key caches to disk
- How often Cassandra saves row caches to disk

Configuring the row_cache_size_in_mb (in the cassandra.yaml configuration file) determines how much space in memory Cassandra allocates to store rows from the most frequently read partitions of the table.

Procedure

Set the table caching property that configures the partition key cache and the row cache.

```
CREATE TABLE users (
  userid text PRIMARY KEY,
  first_name text,
  last_name text,
)
WITH caching = { 'keys' : 'NONE', 'rows_per_partition' : '120' };
```

Tips for efficient cache use

Tuning the row cache in Cassandra 2.1 describes best practices of using the built-in caching mechanisms and designing an effective data model. Some tips for efficient cache use are:

- Store lower-demand data or data with extremely long partitions in a table with minimal or no caching.
- Deploy a large number of Cassandra nodes under a relatively light load per node.
- Logically separate heavily-read data into discrete tables.

When you query a table, turn on tracing to check that the table actually gets data from the cache rather than from disk. The first time you read data from a partition, the trace shows this line below the query because the cache has not been populated yet:

```
Row cache miss [ReadStage:41]
```

In subsequent queries for the same partition, look for a line in the trace that looks something like this:

```
Row cache hit [ReadStage:55]
```

This output means the data was found in the cache and no disk read occurred. Updates invalidate the cache. If you query rows in the cache plus uncached rows, request more rows than the global limit allows, or the query does not grab the beginning of the partition, the trace might include a line that looks something like this:

```
Ignoring row cache as cached value could not satisfy query [ReadStage:89]
```

This output indicates that an insufficient cache caused a disk read. Requesting rows not at the beginning of the partition is a likely cause. Try removing constraints that might cause the query to skip the beginning of the partition, or place a limit on the query to prevent results from overflowing the cache. To ensure that the query hits the cache, try increasing the cache size limit, or restructure the table to position frequently accessed rows at the head of the partition.

Monitoring and adjusting caching

Make changes to cache options in small, incremental adjustments, then monitor the effects of each change using the nodetool utility. The output of the `nodetool info` command shows the following row cache and key cache setting values, which are configured in the cassandra.yaml file:

- Cache size in bytes
- Capacity in bytes
- Number of hits
- Number of requests
- Recent hit rate
- Duration in seconds after which Cassandra saves the key cache.

For example, on start-up, the information from `nodetool info` might look something like this:

```
ID                 : 387d15ba-7103-491b-9327-1a691dbb504a
Gossip active      : true
Thrift active      : true
Native Transport active: true
Load               : 65.87 KB
Generation No      : 1400189757
Uptime (seconds) : 148760
Heap Memory (MB) : 392.82 / 1996.81
datacenter      : datacenter1
Rack               : rack1
Exceptions         : 0
Key Cache          : entries 10, size 728 (bytes), capacity 103809024 (bytes),
 93 hits, 102 requests, 0.912 recent hit rate, 14400 save period in seconds
Row Cache          : entries 0, size 0 (bytes), capacity 0 (bytes), 0 hits, 0
 requests, NaN recent hit rate, 0 save period in seconds
Counter Cache      : entries 0, size 0 (bytes), capacity 51380224 (bytes), 0
 hits, 0 requests, NaN recent hit rate, 7200 save period in seconds
Token              : -9223372036854775808
```

In the event of high memory consumption, consider tuning data caches.

Configuring memtable thresholds

Configuring memtable thresholds can improve write performance. Cassandra flushes memtables to disk, creating SSTables when the commit log space threshold or the memtable cleanup threshold has been exceeded. Configure the commit log space threshold per node in the cassandra.yaml. How you tune memtable thresholds depends on your data and write load. Increase memtable thresholds under either of these conditions:

- The write load includes a high volume of updates on a smaller set of data.
- A steady stream of continuous writes occurs. This action leads to more efficient compaction.

Allocating memory for memtables reduces the memory available for caching and other internal Cassandra structures, so tune carefully and in small increments.

Configuring compaction

As discussed in the Compaction on page 28 topic, the compaction process merges keys, combines columns, evicts tombstones, consolidates SSTables, and creates a new index in the merged SSTable.

In the cassandra.yaml file, you configure these global compaction parameters:

- snapshot_before_compaction
- concurrent_compactors
- compaction_throughput_mb_per_sec

The compaction_throughput_mb_per_sec parameter is designed for use with large partitions. Cassandra throttles compaction to this rate across the entire system.

Cassandra provides a start-up option for testing compaction strategies without affecting the production workload.

Cassandra supports the following compaction strategies, which you can configure using CQL:

- `SizeTieredCompactionStrategy (STCS)`: The default compaction strategy. This strategy triggers a minor compaction when there are a number of similar sized SSTables on disk as configured by the table subproperty, min_threshold. A minor compaction does not involve all the tables in a keyspace. Also see STCS compaction subproperties.
- `DateTieredCompactionStrategy (DTCS)`: This strategy is particularly useful for time series data. DateTieredCompactionStrategy stores data written within a certain period of time in the same SSTable. For example, Cassandra can store your last hour of data in one SSTable *time window*, and the next 4 hours of data in another time window, and so on. Compactions are triggered when the min_threshold (4 by default) for SSTables in those windows is reached. The most common queries for time series workloads retrieve the last hour/day/month of data. Cassandra can limit SSTables returned to those having the relevant data. Also, Cassandra can store data that has been set to expire using TTL in an SSTable with other data scheduled to expire at approximately the same time. Cassandra can then drop the SSTable without doing any compaction. Also see DTCS compaction subproperties and DateTieredCompactionStrategy: Compaction for Time Series Data.
- `TimeWindowCompactionStrategy (TWCS)` This strategy is another alternative for time series data. TWCS compacts SSTables using a series of *time windows* or *buckets*. TWCS creates a new time window within each successive time period. During the active time window, TWCS compacts all SSTables flushed from memory into larger SSTables using STCS. At the end of the time period, all of these SSTables are compacted into a single SSTable. Then the next time window starts and the process repeats. You can configure the duration of the time window. For more information about TWCS, including an example, see How is data maintained? on page 28.
- `LeveledCompactionStrategy (LCS)`: The leveled compaction strategy creates SSTables of a fixed, relatively small size (160 MB by default) that are grouped into levels. Within each level, SSTables are guaranteed to be non-overlapping. Each level (L0, L1, L2 and so on) is 10 times as large as the previous. Disk I/O is more uniform and predictable on higher than on lower levels as SSTables are continuously being compacted into progressively larger levels. At each level, row keys are merged into non-overlapping SSTables in the next level. This process can improve performance for reads, because Cassandra can determine which SSTables in each level to check for the existence of row key data. This compaction strategy is modeled after Google's LevelDB implementation. Also see LCS compaction subproperties.

To configure the compaction strategy property and CQL compaction subproperties, such as the maximum number of SSTables to compact and minimum SSTable size, use CREATE TABLE or ALTER TABLE.

Procedure

1. Update a table to set the compaction strategy using the ALTER TABLE statement.

```
ALTER TABLE users WITH
  compaction = { 'class' :  'LeveledCompactionStrategy'  }
```

2. Change the compaction strategy property to SizeTieredCompactionStrategy and specify the minimum number of SSTables to trigger a compaction using the CQL min_threshold attribute.

```
ALTER TABLE users
  WITH compaction =
  {'class' : 'SizeTieredCompactionStrategy', 'min_threshold' : 6 }
```

Results

You can monitor the results of your configuration using compaction metrics, see Compaction metrics on page 169.

What to do next

Cassandra 3.0 and later support extended logging for Compaction. This utility must be configured as part of the table configuration. The extended compaction logs are stored in a separate file. For details, see Enabling extended compaction logging.

Compression

Compression maximizes the storage capacity of Cassandra nodes by reducing the volume of data on disk and disk I/O, particularly for read-dominated workloads. Cassandra quickly finds the location of rows in the SSTable index and decompresses the relevant row chunks. Compression is important for Cassandra 2.2, but Cassandra 3.0 and later uses a new storage engine that dramatically reduces disk volume automatically. For information on the Cassandra 3.0 improvements, see Putting some structure in the storage engine

Write performance is not negatively impacted by compression in Cassandra as it is in traditional databases. In traditional relational databases, writes require overwrites to existing data files on disk. The database has to locate the relevant pages on disk, decompress them, overwrite the relevant data, and finally recompress. In a relational database, compression is an expensive operation in terms of CPU cycles and disk I/O. Because Cassandra SSTable data files are immutable (they are not written to again after they have been flushed to disk), there is no recompression cycle necessary in order to process writes. SSTables are compressed only once when they are written to disk. Writes on compressed tables can show up to a 10 percent performance improvement.

In Cassandra 2.2 and later, the commit log can also be compressed and write performance can be improved 6-12%. For more information, see Updates to Cassandra’s Commit Log in 2.2.

When to compress data

Compression is most effective on a table with many rows, where each row contains the same set of columns (or the same number of columns) as all other rows. For example, a table containing user data such as *username*, *email* and *state* is a good candidate for compression. The greater the similarity of the data across rows, the greater the compression ratio and gain in read performance.

A table whose rows contain differing sets of columns is not well-suited for compression.

Don't confuse table compression with compact storage of columns, which is used for backward compatibility of old applications with CQL.

Depending on the data characteristics of the table, compressing its data can result in:

- 25-33% reduction in data size
- 25-35% performance improvement on reads
- 5-10% performance improvement on writes

After configuring compression on an existing table, subsequently created SSTables are compressed. Existing SSTables on disk are not compressed immediately. Cassandra compresses existing SSTables when the normal Cassandra compaction process occurs. Force existing SSTables to be rewritten and compressed by using nodetool upgradesstables (Cassandra 1.0.4 or later) or nodetool scrub.

Configuring compression

You configure a table property and subproperties to manage compression. The CQL table properties documentation describes the types of compression options that are available. Compression is enabled by default.

Procedure

1. Disable compression, using CQL to set the compression parameter `enabled` to `false`.

   ```
   CREATE TABLE DogTypes (
                 block_id uuid,
                 species text,
                 alias text,
                 population varint,
                 PRIMARY KEY (block_id)
               )
               WITH compression = { 'enabled' : false };
   ```

2. Enable compression on an existing table, using ALTER TABLE to set the compression algorithm `class` to LZ4Compressor (Cassandra 1.2.2 and later), SnappyCompressor, or DeflateCompressor.

   ```
   CREATE TABLE DogTypes (
                 block_id uuid,
                 species text,
                 alias text,
                 population varint,
                 PRIMARY KEY (block_id)
               )
               WITH compression = { 'class' : 'LZ4Compressor' };
   ```

3. Change compression on an existing table, using ALTER TABLE and setting the compression algorithm `class` to `DeflateCompressor`.

   ```
   ALTER TABLE CatTypes
     WITH compression = { 'class' : 'DeflateCompressor',
    'chunk_length_in_kb' : 64 }
   ```

   You tune data compression on a per-table basis using CQL to alter a table.

Testing compaction and compression

Write survey mode is a Cassandra startup option for testing new compaction and compression strategies. In write survey mode, you can test out new compaction and compression strategies on that node and benchmark the write performance differences, without affecting the production cluster.

Write survey mode adds a node to a database cluster. The node accepts all write traffic as if it were part of the normal Cassandra cluster, but the node does not officially join the ring.

Also use write survey mode to try out a new Cassandra version. The nodes you add in write survey mode to a cluster must be of the same major release version as other nodes in the cluster. The write survey mode relies on the streaming subsystem that transfers data between nodes in bulk and differs from one major release to another.

If you want to see how read performance is affected by modifications, stop the node, bring it up as a standalone machine, and then benchmark read operations on the node.

Procedure

Start the Cassandra node using the write_survey option:

- Package installations: Add the following option to cassandra-env.sh file:

  ```
  JVM_OPTS="$JVM_OPTS -Dcassandra.write_survey=true
  ```

- Tarball installations: Start Cassandra with this option:

  ```
  $ cd install_location
  $ sudo bin/cassandra -Dcassandra.write_survey=true
  ```

Tuning Bloom filters

Cassandra uses Bloom filters to determine whether an SSTable has data for a particular partition. Bloom filters are unused for range scans, but are used for index scans. Bloom filters are probabilistic sets that allow you to trade memory for accuracy. This means that higher Bloom filter attribute settings bloom_filter_fp_chance use less memory, but will result in more disk I/O if the SSTables are highly fragmented. Bloom filter settings range from 0 to 1.0 (disabled). The default value of bloom_filter_fp_chance depends on the compaction strategy. The LeveledCompactionStrategy uses a higher default value (0.1) than the SizeTieredCompactionStrategy or DateTieredCompactionStrategy, which have a default of 0.01. Memory savings are nonlinear; going from 0.01 to 0.1 saves about one third of the memory. SSTables using LCS contain a relatively smaller ranges of keys than those using STCS, which facilitates efficient exclusion of the SSTables even without a bloom filter; however, adding a small bloom filter helps when there are many levels in LCS.

The settings you choose depend the type of workload. For example, to run an analytics application that heavily scans a particular table, you would want to inhibit the Bloom filter on the table by setting it high.

To view the observed Bloom filters false positive rate and the number of SSTables consulted per read use tablestats in the nodetool utility.

Bloom filters are stored off-heap so you don't need include it when determining the -Xmx settings (the maximum memory size that the heap can reach for the JVM).

To change the bloom filter property on a table, use CQL. For example:

```
ALTER TABLE addamsFamily WITH bloom_filter_fp_chance = 0.1;
```

After updating the value of bloom_filter_fp_chance on a table, Bloom filters need to be regenerated in one of these ways:

- Initiate compaction
- Upgrade SSTables

You do not have to restart Cassandra after regenerating SSTables.

Moving data to or from other databases

Cassandra offers several solutions for migrating from other databases:

- The COPY command, which mirrors what the PostgreSQL RDBMS uses for file/export import.
- The Cassandra bulk loader provides the ability to bulk load external data into a cluster.

About the COPY command

You can use COPY in CQL shell to load flat file data into Cassandra (nearly all relational databases have unload utilities that allow table data to be written to OS files) as well to write Cassandra data to CSV files.

ETL Tools

If you need more sophistication applied to a data movement situation (more than just extract-load), then you can use any number of extract-transform-load (ETL) solutions that now support Cassandra. These tools provide excellent transformation routines that allow you to manipulate source data in literally any way you need and then load it into a Cassandra target. They also supply many other features such as visual, point-and-click interfaces, scheduling engines, and more.

Many ETL vendors who support Cassandra supply community editions of their products that are free and able to solve many different use cases. Enterprise editions are also available that supply many other compelling features that serious enterprise data users need.

You can freely download and try ETL tools from Jaspersoft, Pentaho, and Talend that all work with Cassandra.

Purging gossip state on a node

Gossip information is persisted locally by each node to use immediately on node restart without having to wait for gossip communications.

Procedure

In the unlikely event you need to correct a problem in the gossip state:

1. Use the nodetool assassinate to shut down the problem node.

   This takes approximately 35 seconds to complete, so wait for confirmation that the node is deleted.
2. If this method doesn't solve the problem, stop your client application from sending writes to the cluster.
3. Take the entire cluster offline:

   a) Drain each node.

   ```
   $ nodetool options drain
   ```

   b) Stop each node:

   - Package installations:

     ```
     $ sudo service cassandra stop
     ```
   - Tarball installations:

     ```
     $ sudo service cassandra stop
     ```
4. Clear the data from the `peers` directory, remove all directories in the peers-*UUID* directory, where *UUID* is the particular directory that corresponds to the appropriate node:

   ```
   $ sudo rm -r /var/lib/cassandra/data/system/peers-UUID/*
   ```

   CAUTION:

   Use caution when performing this step. The action clears internal system data from Cassandra and may cause application outage without careful execution and validation of the results. To validate the results, run the following query individually on each node to confirm that all of the nodes are able to see all other nodes.

   ```
   select * from system.peers;
   ```
5. Clear the gossip state when the node starts:

   - For tarball installations, you can use a command line option or edit the `cassandra-env.sh`. To use the command line:

     ```
     $ install_location/bin/cassandra -Dcassandra.load_ring_state=false
     ```

- For package installations or if you are not using the command line option above, add the following line to the cassandra-env.sh file:

  ```
  $env:JVM_OPTS="$JVM_OPTS -Dcassandra.load_ring_state=false"
  ```

  - Package installations: `/usr/share/cassandra/cassandra-env.sh`
  - Tarball installations: `install_location/conf/cassandra-env.sh`

6. Bring the cluster online one node at a time, starting with the seed nodes.

   - Package installations:

     ```
     $ sudo service cassandra start
     ```
   - Tarball installations:

     ```
     $ cd install_location
     $ bin/cassandra
     ```

What to do next

Remove the line you added in the `cassandra-env.sh` file.

Cassandra tools

The nodetool utility

About the nodetool utility

The nodetool utility is a command line interface for managing a cluster.

Command formats

```
$ nodetool [options] command [args]
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.

Note:

- For tarball installations, execute the command from the *install_location*/bin directory.
- If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials.

- The repair and rebuild commands can affect multiple nodes in the cluster.
- Most nodetool commands operate on a single node in the cluster if -h is not used to identify one or more other nodes. If the node from which you issue the command is the intended target, you do not need the -h option to identify the target; otherwise, for remote invocation, identify the target node, or nodes, using -h.

Example

```
$ nodetool -u cassandra -pw cassandra describering demo_keyspace
```

Getting nodetool help

`nodetool help`

Provides a listing of nodetool commands.

`nodetool help` *`command name`*

Provides help on a specific command. For example:

```
$ nodetool help upgradesstables
```

For more information, see nodetool help

nodetool assassinate

Forcefully removes a dead node without re-replicating any data. It is a last resort tool if you cannot successfully use nodetool removenode.

Synopsis

```
$ nodetool [options] assassinate <ip_address>
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
ip_address	IP address of the endpoint to assassinate.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Note:

- For tarball installations, execute the command from the *install_location*/bin directory.
- If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials.
- `nodetool assassinate` operates on a single node in the cluster if -h is not used to identify one or more other nodes. If the node from which you issue the command is the intended target, you do not need the -h option to identify the target; otherwise, for remote invocation, identify the target node, or nodes, using -h.

Description

The `nodetool assassinate` command is a tool of last resort. Only use this tool to remove a node from a cluster when `removenode` is not successful.

Examples

```
$ nodetool -u cassandra -pw cassandra assassinate 192.168.100.2
```

nodetool bootstrap

Monitor and manage a node's bootstrap process.

Synopsis

```
$ nodetool [options] bootstrap [resume]
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`--`	Separates an option from an argument that could be mistaken for a option.	

Note:

- For tarball installations, execute the command from the *install_location*/bin directory.
- If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials.
- `nodetool bootstrap` operates on a single node in the cluster if -h is not used to identify one or more other nodes. If the node from which you issue the command is the intended target, you do not need the -h option to identify the target; otherwise, for remote invocation, identify the target node, or nodes, using -h.

Description

The `nodetool bootstrap` command can be used to monitor and manage a node's bootstrap process. If no argument is defined, the help information is displayed. If the argument `resume` is used, bootstrap streaming is resumed.

Examples

```
$ nodetool -u cassandra -pw cassandra bootstrap resume
```

nodetool cfhistograms

This tool has been renamed as tablehistograms.

nodetool cfstats

This tool has been renamed as nodetool tablestats.

nodetool cleanup

Cleans up keyspaces and partition keys no longer belonging to a node.

Synopsis

```
$ nodetool <options> cleanup --  <keyspace> (<table> ...)
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
keyspace	Keyspace name.	
table	One or more table names, separated by a space.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

Use this command to remove unwanted data after adding a new node to the cluster. Cassandra does not automatically remove data from nodes that lose part of their partition range to a newly added node. Run `nodetool cleanup` on the source node and on neighboring nodes that shared the same subrange after the new node is up and running. Failure to run this command after adding a node causes Cassandra to include the old data to rebalance the load on that node. Running the `nodetool cleanup` command causes a temporary increase in disk space usage proportional to the size of your largest SSTable. Disk I/O occurs when running this command.

Running this command affects nodes that use a counter column in a table. Cassandra assigns a new counter ID to the node.

Optionally, this command takes a list of table names. If you do not specify a keyspace, this command cleans all keyspaces no longer belonging to a node.

nodetool clearsnapshot

Removes one or more snapshots.

Synopsis

```
$ nodetool <options> clearsnapshot -t <snapshot> -- ( <keyspace> ... )
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
-t	Remove the snapshot with a designated name.	
keyspace	Remove snapshots from the designated keyspaces, separated by a space.	
snapshot	Name of the snapshot.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

Deletes snapshots in one or more keyspaces. To remove all snapshots, omit the snapshot name.

nodetool compact

Forces a major compaction on one or more tables.

Synopsis

```
$ nodetool <options> compact <keyspace> ( <table> ... )
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Remote JMX agent port number.
`-pw`	`--password`	Password.
`-pwf`	`--password-file`	Password file path.
`-s`	--split-output	Split output of STCS files to 50%-25%-12.5% and so on of the total size.
`-u`	`--username`	Remote JMX agent user name.
--user-defined	Submit listed files for user-defined compaction. For Cassandra 3.4 and later.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Note:

- For tarball installations, execute the command from the *install_location*/bin directory.
- If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials.
- No `-s` will create one large SSTable for STCS.
- `-s` will not affect DTCS; it will create one large SSTable.

Description

This command starts the compaction process on tables using SizeTieredCompactionStrategy (STCS), DateTieredCompactionStrategy (DTCS), or Leveled compaction (LCS):

- If you do not specify a keyspace or table, a major compaction is run on all keyspaces and tables.
- If you specify only a keyspace, a major compaction is run on all tables in that keyspace.
- If you specify one or more tables, a major compaction is run on those tables.

Major compactions may behave differently depending which compaction strategy is used for the affected tables:

- `SizeTieredCompactionStrategy (STCS)`: The default compaction strategy. This strategy triggers a minor compaction when there are a number of similar sized SSTables on disk as configured by the table subproperty, min_threshold. A minor compaction does not involve all the tables in a keyspace. Also see STCS compaction subproperties.
- `DateTieredCompactionStrategy (DTCS)`: This strategy is particularly useful for time series data. DateTieredCompactionStrategy stores data written within a certain period of time in the same SSTable. For example, Cassandra can store your last hour of data in one SSTable *time window*, and the next 4 hours of data in another time window, and so on. Compactions are triggered when the min_threshold (4 by default) for SSTables in those windows is reached. The most common queries for time series workloads retrieve the last hour/day/month of data. Cassandra can limit SSTables returned to those having the relevant data. Also, Cassandra can store data that has been set to expire using TTL in an SSTable with other data scheduled to expire at approximately the same time. Cassandra can then drop the SSTable without doing any compaction. Also see DTCS compaction subproperties and DateTieredCompactionStrategy: Compaction for Time Series Data.
- `TimwWindowCompactionStrategy (TWCS)` This strategy is another alternative for time series data. TWCS compacts SSTables using a series of *time windows* or *buckets*. TWCS creates a new time window within each successive time period. During the active time window, TWCS compacts all SSTables flushed from memory into larger SSTables using STCS. At the end of the time period, all of these SSTables are compacted into a single SSTable. Then the next time window starts and the process repeats. You can configure the duration of the time window. For more information about TWCS, including an example, see How is data maintained? on page 28.
- `TimeWindowCompactionStrategy (TWCS)` This strategy is another alternative for time series data. TWCS compacts SSTables using a series of *time windows* or *buckets*. TWCS creates a new time window within each successive time period. During the active time window, TWCS compacts all SSTables flushed from memory into larger SSTables using STCS. At the end of the time period, all of these SSTables are compacted into a single SSTable. Then the next time window starts and the process repeats. You can configure the duration of the time window. For more information about TWCS, including an example, see How is data maintained? on page 28.
- `LeveledCompactionStrategy (LCS)`: The leveled compaction strategy creates SSTables of a fixed, relatively small size (160 MB by default) that are grouped into levels. Within each level, SSTables are guaranteed to be non-overlapping. Each level (L0, L1, L2 and so on) is 10 times as large as the previous. Disk I/O is more uniform and predictable on higher than on lower levels as SSTables are continuously being compacted into progressively larger levels. At each level, row keys are merged into non-overlapping SSTables in the next level. This process can improve performance for reads, because Cassandra can determine which SSTables in each level to check for the existence of row key data. This compaction strategy is modeled after Google's LevelDB implementation. Also see LCS compaction subproperties.

For more details, see How is data maintained? and Configuring compaction.

Note: A major compaction incurs considerably more disk I/O than minor compactions.

nodetool compactionhistory

Provides the history of compaction operations.

Synopsis

```
$ nodetool <options> compactionhistory
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Example

The actual output of compaction history is seven columns wide. The first three columns show the id, keyspace name, and table name of the compacted SSTable.

```
$ nodetool compactionhistory
```

```
Compaction History:
id                                       keyspace_name
 columnfamily_name
d06f7080-07a5-11e4-9b36-abc3a0ec9088     system
 schema_columnfamilies
d198ae40-07a5-11e4-9b36-abc3a0ec9088     libdata            users
0381bc30-07b0-11e4-9b36-abc3a0ec9088     Keyspace1          Standard1
74eb69b0-0621-11e4-9b36-abc3a0ec9088     system             local
e35dd980-07ae-11e4-9b36-abc3a0ec9088     system
 compactions_in_progress
8d5cf160-07ae-11e4-9b36-abc3a0ec9088     system
 compactions_in_progress
ba376020-07af-11e4-9b36-abc3a0ec9088     Keyspace1          Standard1
d18cc760-07a5-11e4-9b36-abc3a0ec9088     libdata            libout
64009bf0-07a4-11e4-9b36-abc3a0ec9088     libdata            libout
d04700f0-07a5-11e4-9b36-abc3a0ec9088     system             sstable_activity
c2a97370-07a9-11e4-9b36-abc3a0ec9088     libdata            users
cb928a80-07ae-11e4-9b36-abc3a0ec9088     Keyspace1          Standard1
cd8d1540-079e-11e4-9b36-abc3a0ec9088     system             schema_columns
62ced2b0-07a4-11e4-9b36-abc3a0ec9088     system             schema_keyspaces
d19cccf0-07a5-11e4-9b36-abc3a0ec9088     system
 compactions_in_progress
640bbf80-07a4-11e4-9b36-abc3a0ec9088     libdata            users
6cd54e60-07ae-11e4-9b36-abc3a0ec9088     Keyspace1          Standard1
c29241f0-07a9-11e4-9b36-abc3a0ec9088     libdata            libout
c2a30ad0-07a9-11e4-9b36-abc3a0ec9088     system
 compactions_in_progress
e3a6d920-079d-11e4-9b36-abc3a0ec9088     system             schema_keyspaces
62c55cd0-07a4-11e4-9b36-abc3a0ec9088     system
 schema_columnfamilies
62b07540-07a4-11e4-9b36-abc3a0ec9088     system             schema_columns
cdd038c0-079e-11e4-9b36-abc3a0ec9088     system             schema_keyspaces
b797af00-07af-11e4-9b36-abc3a0ec9088     Keyspace1          Standard1
8c918b10-07ae-11e4-9b36-abc3a0ec9088     Keyspace1          Standard1
377d73f0-07ae-11e4-9b36-abc3a0ec9088     system
 compactions_in_progress
```

```
62b9c410-07a4-11e4-9b36-abc3a0ec9088     system             local
d0566a40-07a5-11e4-9b36-abc3a0ec9088     system             schema_columns
ba637930-07af-11e4-9b36-abc3a0ec9088     system
 compactions_in_progress
cdbc1480-079e-11e4-9b36-abc3a0ec9088     system
 schema_columnfamilies
e3456f80-07ae-11e4-9b36-abc3a0ec9088     Keyspace1          Standard1
d086f020-07a5-11e4-9b36-abc3a0ec9088     system             schema_keyspaces
d06118a0-07a5-11e4-9b36-abc3a0ec9088     system             local
cdaafd80-079e-11e4-9b36-abc3a0ec9088     system             local
640fde30-07a4-11e4-9b36-abc3a0ec9088     system
 compactions_in_progress
37638350-07ae-11e4-9b36-abc3a0ec9088     Keyspace1          Standard1
```

The four columns to the right of the table name show the timestamp, size of the SSTable before and after compaction, and the number of partitions merged. The notation means {tables:rows}. For example: {1:3, 3:1} means 3 rows were taken from one SSTable (1:3) and 1 row taken from 3 SSTables (3:1) to make the one SSTable in that compaction operation.

```

. . . compacted_at        bytes_in       bytes_out      rows_merged
. . . 1404936947592       8096           7211           {1:3, 3:1}
. . . 1404936949540       144            144            {1:1}
. . . 1404941328243       1305838191     1305838191     {1:4647111}
. . . 1404770149323       5864           5701           {4:1}
. . . 1404940844824       573            148            {1:1, 2:2}
. . . 1404940700534       576            155            {1:1, 2:2}
. . . 1404941205282       766331398      766331398      {1:2727158}
. . . 1404936949462       8901649        8901649        {1:9315}
. . . 1404936336175       8900821        8900821        {1:9315}
. . . 1404936947327       223            108            {1:3, 2:1}
. . . 1404938642471       144            144            {1:1}
. . . 1404940804904       383020422      383020422      {1:1363062}
. . . 1404933936276       4889           4177           {1:4}
. . . 1404936334171       441            281            {1:3, 2:1}
. . . 1404936949567       379            79             {2:2}
. . . 1404936336248       144            144            {1:1}
. . . 1404940645958       307520780      307520780      {1:1094380}
. . . 1404938642319       8901649        8901649        {1:9315}
. . . 1404938642429       416            165            {1:3, 2:1}
. . . 1404933543858       692            281            {1:3, 2:1}
. . . 1404936334109       7760           7186           {1:3, 2:1}
. . . 1404936333972       4860           4724           {1:2, 2:1}
. . . 1404933936715       441            281            {1:3, 2:1}
. . . 1404941200880       1269180898     1003196133     {1:2623528,
 2:946565}
. . . 1404940699201       297639696      297639696      {1:1059216}
. . . 1404940556463       592            148            {1:2, 2:2}
. . . 1404936334033       5760           5680           {2:1}
. . . 1404936947428       8413           5316           {1:2, 3:1}
. . . 1404941205571       429            42             {2:2}
. . . 1404933936584       7994           6789           {1:4}
. . . 1404940844664       306699417      306699417      {1:1091457}
. . . 1404936947746       601            281            {1:3, 3:1}
. . . 1404936947498       5840           5680           {3:1}
. . . 1404933936472       5861           5680           {3:1}
. . . 1404936336275       378            80             {2:2}
. . . 1404940556293       302170540      281000000      {1:924660, 2:75340}
```

nodetool compactionstats

Provide statistics about a compaction.

Synopsis

```
$ nodetool <options> compactionstats -H
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
-H	--human-readable	Display bytes in human readable form: KiB (kibibyte), MiB (mebibyte), GiB (gibibyte), TiB (tebibyte).

Description

The total column shows the total number of uncompressed bytes of SSTables being compacted. The system log lists the names of the SSTables compacted.

Example

```
$ nodetool compactionstats
```

```
pending tasks: 5
          compaction type        keyspace           table       completed
        total      unit  progress
               Compaction       Keyspace1       Standard1       282310680
    302170540     bytes    93.43%
               Compaction       Keyspace1       Standard1        58457931
    307520780     bytes    19.01%
Active compaction remaining time :   0h00m16s
```

nodetool decommission

Deactivates a node by streaming its data to another node.

Synopsis

```
$ nodetool <options> decommission
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.

Short	Long	Description
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Description

Causes a live node to decommission itself, streaming its data to the next node on the ring. Use netstats to monitor the progress, as described on http://wiki.apache.org/cassandra/NodeProbe#Decommission and http://wiki.apache.org/cassandra/Operations#Removing_nodes_entirely.

nodetool describecluster

Provide the name, snitch, partitioner and schema version of a cluster

Synopsis

```
$ nodetool <options> describecluster -- <datacenter>
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Description

Describe cluster is typically used to validate the schema after upgrading. If a schema disagreement occurs, check for and resolve schema disagreements.

Example

```
$ nodetool describecluster
```

```
Cluster Information:
 Name: Test Cluster
 Snitch: org.apache.cassandra.locator.DynamicEndpointSnitch
 Partitioner: org.apache.cassandra.dht.Murmur3Partitioner
 Schema versions:
  65e78f0e-e81e-30d8-a631-a65dff93bf82: [127.0.0.1]
```

If a schema disagreement occurs, the last line of the output includes information about unreachable nodes.

```
$ nodetool describecluster
```

```
Cluster Information:
 Name: Production Cluster
       Snitch: org.apache.cassandra.locator.DynamicEndpointSnitch
       Partitioner: org.apache.cassandra.dht.Murmur3Partitioner
       Schema versions:
              UNREACHABLE: 1176b7ac-8993-395d-85fd-41b89ef49fbb:
 [10.202.205.203]
```

nodetool describering

Provides the partition ranges of a keyspace.

Synopsis

```
$ nodetool <options> describering -- <keyspace>
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
keyspace	Name of keyspace.	
--	Separates an option from an argument that could be mistaken for a option.	

Example

This example shows the sample output of the command on a three-node cluster.

```
$ nodetool describering demo_keyspace
```

```
Schema Version:1b04bd14-0324-3fc8-8bcb-9256d1e15f82
TokenRange:
 TokenRange(start_token:3074457345618258602,
 end_token:-9223372036854775808,
         endpoints:[127.0.0.1, 127.0.0.2, 127.0.0.3],
         rpc_endpoints:[127.0.0.1, 127.0.0.2, 127.0.0.3],
         endpoint_details:[EndpointDetails(host:127.0.0.1,
 datacenter:datacenter1, rack:rack1),
         EndpointDetails(host:127.0.0.2, datacenter:datacenter1,
 rack:rack1),
         EndpointDetails(host:127.0.0.3, datacenter:datacenter1,
 rack:rack1)])
 TokenRange(start_token:-3074457345618258603,
 end_token:3074457345618258602,
         endpoints:[127.0.0.3, 127.0.0.1, 127.0.0.2],
         rpc_endpoints:[127.0.0.3, 127.0.0.1, 127.0.0.2],
         endpoint_details:[EndpointDetails(host:127.0.0.3,
         datacenter:datacenter1, rack:rack1),
         EndpointDetails(host:127.0.0.1, datacenter:datacenter1,
 rack:rack1),
         EndpointDetails(host:127.0.0.2, datacenter:datacenter1,
 rack:rack1)])
 TokenRange(start_token:-9223372036854775808,
 end_token:-3074457345618258603,
         endpoints:[127.0.0.2, 127.0.0.3, 127.0.0.1],
         rpc_endpoints:[127.0.0.2, 127.0.0.3, 127.0.0.1],
         endpoint_details:[EndpointDetails(host:127.0.0.2,
 datacenter:datacenter1, rack:rack1),
         EndpointDetails(host:127.0.0.3, datacenter:datacenter1,
 rack:rack1),
         EndpointDetails(host:127.0.0.1, datacenter:datacenter1,
 rack:rack1)])
```

If a schema disagreement occurs, the last line of the output includes information about unreachable nodes.

```
$ nodetool describecluster
```

```
Cluster Information:
 Name: Production Cluster
       Snitch: org.apache.cassandra.locator.DynamicEndpointSnitch
       Partitioner: org.apache.cassandra.dht.Murmur3Partitioner
       Schema versions:
              UNREACHABLE: 1176b7ac-8993-395d-85fd-41b89ef49fbb:
 [10.202.205.203]
```

nodetool disableautocompaction

Disables autocompaction for a keyspace and one or more tables.

Synopsis

```
$ nodetool <options> disableautocompaction -- <keyspace> ( <table> ... )
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
keyspace	Keyspace name.	
table	One or more table names, separated by a space.	
--	Separates an option from an argument that could be mistaken for a option.	

Description

The keyspace can be followed by one or more tables.

nodetool disablebackup

Disables incremental backup.

Synopsis

```
$ nodetool <options> disablebackup
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

nodetool disablebinary

Disables the native transport.

Synopsis

```
$ nodetool <options> disablebinary
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Description

Disables the binary protocol, also known as the native transport.

nodetool disablegossip

Disables the gossip protocol.

Synopsis

```
$ nodetool <options> disablegossip
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Description

This command effectively marks the node as being down.

nodetool disablehandoff

Disables storing of future hints on the current node.

Synopsis

```
$ nodetool <options> disablehandoff
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

nodetool disablehintsfordc

Disable hints for a datacenter.

Synopsis

```
$ nodetool [options] disablehintsfordc [--] <datacenter>
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.

Short	Long	Description
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
datacenter	The data center to disable.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Note:

- For tarball installations, execute the command from the *install_location*/bin directory.
- If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials.
- `nodetool disablehintsfordc` operates on a single node in the cluster if -h is not used to identify one or more other nodes. If the node from which you issue the command is the intended target, you do not need the -h option to identify the target; otherwise, for remote invocation, identify the target node, or nodes, using -h.
- [--] can be used to separate command-line options from the list of arguments, when the list might be mistaken for options.

Description

The `nodetool disablehintsfordc` command is used to turn off hints for a datacenter. This can be useful if there is a downed datacenter, but hints should continue on other datacenters. Another common case is during datacenter failover, when hints will put unnecessary pressure on the datacenter.

Examples

```
$ nodetool -u cassandra -pw cassandra disablehintsfordc DC2
```

nodetool disablethrift

Disables the Thrift server.

Synopsis

```
$ nodetool [options] disablethrift
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`--`	Separates an option from an argument that could be mistaken for a option.	

Note:

- For tarball installations, execute the command from the *install_location*/bin directory.
- If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials.
- `nodetool disablethrift` operates on a single node in the cluster if -h is not used to identify one or more other nodes. If the node from which you issue the command is the intended target, you do not need the -h option to identify the target; otherwise, for remote invocation, identify the target node, or nodes, using -h.

Description

`nodetool disablethrift` will disable thrift on a node preventing the node from acting as a coordinator. The node can still be a replica for a different coordinator and data read at consistency level ONE could be stale. To cause a node to ignore read requests from other coordinators, `nodetool disablegossip` would also need to be run. However, if both commands are run, the node will not perform repairs, and the node will continue to store stale data. If the goal is to repair the node, set the read operations to a consistency level of QUORUM or higher while you run repair. An alternative approach is to delete the node's data and restart the Cassandra process.

Note that the `nodetool` commands using the `-h` option will not work remotely on a disabled node until `nodetool enablethrift` and `nodetool enablegossip` are run locally on the disabled node.

Examples

```
$ nodetool -u cassandra -pw cassandra disablethrift 192.168.100.1
```

nodetool drain

Drains the node.

Synopsis

```
$ nodetool <options> drain
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Description

Flushes all memtables from the node to SSTables on disk. Cassandra stops listening for connections from the client and other nodes. You need to restart Cassandra after running `nodetool drain`. You typically use this command before upgrading a node to a new version of Cassandra. To simply flush memtables to disk, use `nodetool flush`.

nodetool enableautocompaction

Enables autocompaction for a keyspace and one or more tables.

Synopsis

```
$ nodetool <options> enableautocompaction -- <keyspace> ( <table> ... )
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
keyspace	Name of keyspace.	
table	One or more table names, separated by a space.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

The keyspace can be followed by one or more tables. Enables compaction for the named keyspace or the current keyspace, and one or more named tables, or all tables.

nodetool enablebackup

Enables incremental backup.

Synopsis

```
$ nodetool <options> enablebackup
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`--`	Separates an option from an argument that could be mistaken for a option.	

nodetool enablebinary

Re-enables native transport.

Synopsis

```
$ nodetool <options> enablebinary
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Description

Re-enables the binary protocol, also known as native transport.

nodetool enablegossip

Re-enables gossip.

Synopsis

```
$ nodetool <options> enablegossip
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

nodetool enablehandoff

Re-enables the storing of future hints on the current node.

Synopsis

```
$ nodetool <options> enablehandoff
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`--`	Separates an option from an argument that could be mistaken for a option.	

nodetool enablehintsfordc

Enable hints for a datacenter.

Synopsis

```
$ nodetool [options] enablehintsfordc [--] <datacenter>
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
datacenter	The datacenter to enable.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Note:

- For tarball installations, execute the command from the *install_location*/bin directory.
- If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials.
- `nodetool enablehintsfordc` operates on a single node in the cluster if -h is not used to identify one or more other nodes. If the node from which you issue the command is the intended target, you do not need the -h option to identify the target; otherwise, for remote invocation, identify the target node, or nodes, using -h.
- [--] can be used to separate command-line options from the list of arguments, when the list might be mistaken for options.

Description

The `nodetool enablehintsfordc` command is used to turn on hints for a datacenter. The cassandra.yaml file has a parameter, hinted_handoff_disabled_datacenters that will blacklist datacenters on startup. If a datacenter can be enabled later with `nodetool enablehintsfordc`.

Examples

```
$ nodetool -u cassandra -pw cassandra enablehintsfordc DC2
```

nodetool enablethrift

Re-enables the Thrift server.

Synopsis

```
$ nodetool <options> enablethrift
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

nodetool failuredetector

Shows the failure detector information for the cluster.

Synopsis

```
$ nodetool [(-h <host> | --host <host>)] [(-p <port> | --port <port>)]
     [(-pw <password> | --password <password>)]
     [(-pwf <passwordFilePath> | --password-file <passwordFilePath>)]
     [(-u <username> | --username <username>)] failuredetector
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Description

Shows the failure detector information for the cluster.

nodetool flush

Flushes one or more tables from the memtable.

Synopsis

```
$ nodetool <options> flush -- <keyspace> ( <table> ... )
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
keyspace	Name of keyspace.	
table	One or more table names, separated by a space.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

You can specify a keyspace followed by one or more tables that you want to flush from the memtable to SSTables on disk.

nodetool gcstats

Print garbage collection (GC) statistics.

Synopsis

```
$ nodetool [options] gcstats
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`--`	Separates an option from an argument that could be mistaken for a option.	

Note:

- For tarball installations, execute the command from the *install_location*/bin directory.

- If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials.
- `nodetool gcstats` operates on a single node in the cluster if -h is not used to identify one or more other nodes. If the node from which you issue the command is the intended target, you do not need the -h option to identify the target; otherwise, for remote invocation, identify the target node, or nodes, using -h.

Description

The `nodetool gcstats` command will print garbage collection statistics that returns values based on all the garbage collection that has run since the last time `nodetool gcstats` was run. Statistics identify the interval time, some GC elapsed time measures, the disk space reclaimed (in MB), number of garbage collections that took place, and direct memory bytes.

Examples

```
$ nodetool -u cassandra -pw cassandra gcstats
```

nodetool getcompactionthreshold

Provides the minimum and maximum compaction thresholds in megabytes for a table.

Synopsis

```
$ nodetool <options> getcompactionthreshold -- <keyspace> <table>
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
keyspace	Name of keyspace.	
table	Name of table.	
`--`	Separates an option from an argument that could be mistaken for a option.	

nodetool getcompactionthroughput

Print the throughput cap (in MB/s) for compaction in the system.

Synopsis

```
$ nodetool [options] getcompactionthroughput
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`--`	Separates an option from an argument that could be mistaken for a option.	

Note:

- For tarball installations, execute the command from the *install_location*/bin directory.
- If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials.
- `nodetool getcompactionthroughput` operates on a single node in the cluster if -h is not used to identify one or more other nodes. If the node from which you issue the command is the intended target, you do not need the -h option to identify the target; otherwise, for remote invocation, identify the target node, or nodes, using -h.

Description

The `nodetool getcompactionthroughput` command prints the current compaction throughput.

Examples

```
$ nodetool -u cassandra -pw cassandra getcompactionthroughput
```

nodetool getendpoints

Provides the IP addresses or names of replicas that own the partition key.

Synopsis

```
$ nodetool <options> getendpoints -- <keyspace> <table> key
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
keyspace	Name of keyspace.	
table	Name of table.	

Short	Long	Description
key	Partition key of the end points you want to get.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Example

For example, which nodes own partition key_1, key_2, and key_3?

Note: The partitioner returns a token for the key. Cassandra will return an endpoint whether or not data exists on the identified node for that token.

```
$ nodetool -h 127.0.0.1 -p 7100 getendpoints myks mytable key_1
```

```
127.0.0.2
```

```
$ nodetool -h 127.0.0.1 -p 7100 getendpoints myks mytable key_2
```

```
127.0.0.2
```

```
$ nodetool -h 127.0.0.1 -p 7100 getendpoints myks mytable key_2
```

```
127.0.0.1
```

nodetool getlogginglevels

Get the runtime logging levels.

Synopsis

```
$ nodetool <options> getlogginglevels
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`--`	Separates an option from an argument that could be mistaken for a option.	

nodetool getsstables

Provides the SSTables that own the partition key.

Synopsis

```
$ nodetool <options> getsstables [(-hf | --hex-format)] -- <keyspace> <table>
 <key>
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
keyspace	Name of keyspace.	
table	One or more table names, separated by a space.	
key	Partition key of the SSTables.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

This command can be used to retrieve an SSTable.

Examples

An example of this command retrieves the SSTable for `cycling.cyclist_name` with the key argument `fb372533-eb95-4bb4-8685-6ef61e994caa` for one of the cyclists listed:

```
$ nodetool getsstables cycling cyclist_name 'fb372533-
eb95-4bb4-8685-6ef61e994caa'
```

The output is:

```
/homedir/datastax-ddc-3.6.0/data/data/cycling/
cyclist_name-612a64002ec211e6a92457e568fce26f/ma-1-big-Data.db
```

Sometimes it's useful to retrieve an SSTable from the hex string representation of its key, for instance, when you get this exception and you want to find out which SSTable owns the faulty key:

```
java.lang.AssertionError: row DecoratedKey(2769066505137675224,
 00040000002e00000800000153441a3ef000) received out of order wrt
 DecoratedKey(2774747040849866654, 0004000019b0000080000015348847eb200)
```

The `nodetool getsstables` command will only work if the primary key of the given table is a blob.

```
nodetool getsstables keyspace table 00040000002e00000800000153441a3ef000
```

For such cases in Cassandra 3.6 and later, the option `--hex-key` can be used to retrieve the DecoratedKey from the hexstr representation of the key:

```
nodetool getsstables ks cf --hex-key 00040000002e00000800000153441a3ef000
```

```
$ nodetool getsstables keyspace1 standard1 3330394c344e35313730
```

nodetool getstreamthroughput

Provides the Mb (megabit) per second outbound throughput limit for streaming in the system.

Synopsis

```
$ nodetool <options> getstreamthroughput
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

nodetool gettimeout

Print the timeout value of the given type in milliseconds (Cassandra 3.4 and later).

Synopsis

```
$ nodetool [options] gettimeout [--] <timeout_type>
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
*timeout*_type	The timeout type, one of read, range, write, counterwrite, cascontention, truncate, streamingsocket, misc (general rpc_timeout_in_ms).	
--	Separates an option from an argument that could be mistaken for a option.	

Note:

- For tarball installations, execute the command from the *install_location*/bin directory.
- If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials.
- -- separates an option and argument that could be mistaken for a option.
- The timeout type:

- read
- range
- write
- counterwrite
- cascontention
- truncate
- streamingsocket
- misc, such as general rpc_timeout_in_ms

Description

The `nodetool gettimeout` command prints the timeout value of the given type in milliseconds. Several timeouts are available.

Examples

```
$ nodetool -u cassandra -pw cassandra gettimeout read
```

nodetool gettraceprobability

Get the current trace probability.

Synopsis

```
$ nodetool <options> gettraceprobability
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

Provides the current trace probability. To set the trace probability, see nodetool settraceprobability.

nodetool gossipinfo

Provides the gossip information for the cluster.

Synopsis

```
$ nodetool <options> gossipinfo
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`--`	Separates an option from an argument that could be mistaken for a option.	

nodetool help

Provides nodetool command help.

Synopsis

```
$ nodetool help <command>
```

Description

The help command provides a synopsis and brief description of each nodetool command.

Examples

Using nodetool help lists all commands and usage information. For example, `nodetool help netstats` provides the following information.

```
NAME
        nodetool netstats - Print network information on provided host
        (connecting node by default)

SYNOPSIS
        nodetool [(-h <host> | --host <host>)] [(-p <port> | --port <port>)]
                [(-pw <password> | --password <password>)]
                [(-u <username> | --username <username>)] netstats

OPTIONS
        -h <host>, --host <host>
            Node hostname or ip address

        -p <port>, --port <port>
            Remote jmx agent port number

        -pw <password>, --password <password>
            Remote jmx agent password

        -u <username>, --username <username>
            Remote jmx agent username
```

nodetool info

Provides node information, such as load and uptime.

Synopsis

```
$ nodetool <options> info ( -T | --tokens )
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
-T	--tokens	Show all tokens.
--	Separates an option from an argument that could be mistaken for a option.	

Description

Provides node information including the token and on disk storage (load) information, times started (generation), uptime in seconds, and heap memory usage.

nodetool invalidatecountercache

Invalidates the counter cache, and resets the global counter cache parameter, counter_cache_keys_to_save, to the default (not set), which saves all keys..

Synopsis

```
$ nodetool [options] invalidatecountercache
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Note:

- For tarball installations, execute the command from the *install_location*/bin directory.
- If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials.
- `nodetool invalidatecountercache` operates on a single node in the cluster if -h is not used to identify one or more other nodes. If the node from which you issue the command is the intended target, you do not need the -h option to identify the target; otherwise, for remote invocation, identify the target node, or nodes, using -h.

Description

The `nodetool invalidatecountercache` command will invalidate the counter cache, and the system will start saving all counter keys.

Examples

```
$ nodetool -u cassandra -pw cassandra invalidatecountercache
```

nodetool invalidatekeycache

Resets the global key cache parameter to the default, which saves all keys.

Synopsis

```
$ nodetool <options> invalidatekeycache
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Description

By default the key_cache_keys_to_save is disabled in the cassandra.yaml. This command resets the parameter to the default.

nodetool invalidaterowcache

Resets the global key cache parameter, row_cache_keys_to_save, to the default (not set), which saves all keys.

Synopsis

```
$ nodetool <options> invalidaterowcache
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.

Short	Long	Description
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

nodetool join

Causes the node to join the ring.

Synopsis

```
$ nodetool <options> join
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Description

Causes the node to join the ring, assuming the node was initially *not* started in the ring using the -Djoin_ring=false cassandra utility option. The joining node should be properly configured with the desired options for seed list, initial token, and auto-bootstrapping.

nodetool listsnapshots

Lists snapshot names, size on disk, and true size.

Synopsis

```
$ nodetool <options> listsnapshots
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Example

```
Snapshot Details:
Snapshot Name  Keyspace   Column Family  True Size   Size on Disk

1387304478196  Keyspace1  Standard1      0 bytes     308.66 MB
1387304417755  Keyspace1  Standard1      0 bytes     107.21 MB
1387305820866  Keyspace1  Standard2      0 bytes      41.69 MB

               Keyspace1  Standard1      0 bytes     308.66 MB
```

nodetool move

Moves the node on the token ring to a new token.

Synopsis

```
$ nodetool <options> move -- <new token>
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
new token	Number in partition range. For Murmur3Partitioner (default): -2^{63} to +2^{63}-1.	
--	Separates an option from an argument that could be mistaken for a option.	

Description

Escape negative tokens using \\ . For example: move \\-123. This command moves a node from one token value to another. This command is generally used to shift tokens slightly.

nodetool netstats

Provides network information about the host.

Synopsis

```
$ nodetool <options> netstats -H
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.

Short	Long	Description
-pw	--password	Password.
-u	--username	Remote JMX agent username.
_H	--human-readable	Display bytes in human readable form: KiB (kibibyte), MiB (mebibyte), GiB (gibibyte), TiB (tebibyte).
--	Separates an option from an argument that could be mistaken for a option.	

Description

The default host is the connected host if the user does not include a host name or IP address in the command. The output includes the following information:

- JVM settings
- Mode

  The operational mode of the node: JOINING, LEAVING, NORMAL, DECOMMISSIONED, CLIENT
- Read repair statistics
- Attempted

  The number of successfully completed read repair operations
- Mismatch (blocking)

  The number of read repair operations since server restart that blocked a query.
- Mismatch (background)

  The number of read repair operations since server restart performed in the background.
- Pool name

  Information about client read and write requests by thread pool.
- Active, pending, and completed number of commands and responses

Example

Get the network information for a node 10.171.147.128:

```
$ nodetool -h 10.171.147.128 netstats
```

The output is:

```
Mode: NORMAL
Not sending any streams.
Read Repair Statistics:
Attempted: 0
Mismatch (Blocking): 0
Mismatch (Background): 0
Pool Name                    Active   Pending      Completed
Commands                        n/a         0           1156
Responses                       n/a         0           2750
```

nodetool pausehandoff

Pauses the hints delivery process

Synopsis

```
$ nodetool <options> pausehandoff
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

nodetool proxyhistograms

Provides a histogram of network statistics at the time of the command.

Synopsis

```
$ nodetool <options> proxyhistograms
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Description

The output of this command shows the full request latency recorded by the coordinator. The output includes the percentile rank of read and write latency values for inter-node communication. Typically, you use the command to see if requests encounter a slow node.

Examples

This example shows the output from nodetool proxyhistograms after running 4,500 insert statements and 45,000 select statements on a three ccm node-cluster on a local computer.

```
$ nodetool proxyhistograms
```

```
proxy histograms
Percentile       Read Latency      Write Latency      Range Latency
                     (micros)           (micros)           (micros)
50%                   1502.50             375.00             446.00
```

```
75%                  1714.75            420.00            498.00
95%                 31210.25            507.00            800.20
98%                 36365.00            577.36            948.40
99%                 36365.00            740.60           1024.39
Min                   616.00            230.00            311.00
Max                 36365.00          55726.00          59247.00
```

In Cassandra 3.6 and later, three metrics have been added to the output:

- CAS Read Latency
- CAS Write Latency
- View Write Latency

CAS Read and Write Latency provides data for Cassandra compare-and-set operations, while View Write Latency provides data for materialized view write operations. The results are slightly different from previous versions:

```
proxy histograms
Percentile       Read Latency      Write Latency      Range Latency   CAS
 Read Latency  CAS Write Latency View Write Latency
                     (micros)           (micros)           (micros)
  (micros)           (micros)           (micros)
50%                    454.83            379.02           1955.67
      0.00               0.00              0.00
75%                   1358.10            943.13           4055.27
      0.00               0.00              0.00
95%                   3379.39          12108.97          20924.30
      0.00               0.00              0.00
98%                   7007.51         155469.30          89970.66
      0.00               0.00              0.00
99%                   8409.01         155469.30         155469.30
      0.00               0.00              0.00
Min                     73.46            126.94            126.94
      0.00               0.00              0.00
Max                  14530.76         155469.30         155469.30
      0.00               0.00              0.00
```

nodetool rangekeysample

Provides the sampled keys held across all keyspaces.

Synopsis

```
$ nodetool <options> rangekeysample
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

nodetool rebuild

Rebuilds data by streaming from other nodes.

Synopsis

```
$ nodetool options rebuild
[ -ks | --keyspace keyspace_name [ , keyspace_name ] . . . ]
[ -ts | --tokens token_spec ]
[ -- source-dc-name ]
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-ks` *specific_keyspace*	`--keyspace` *specific_keyspace*	Rebuild specific keyspace.
source-dc-name	Name of datacenter from which to select sources for streaming. By default, choose any datacenter.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Parameters

***keyspace_name* (Cassandra 3.6 and later)**

The name of the keyspace to rebuild. The `rebuild` command works with a single keyspace or a comma-delimited list of keyspaces.

***token_spec* (Cassandra 3.6 and later)**

One or more tokens, and/or one or more ranges of tokens. The token_spec can be:

- One specific token.
- A comma-delimited list of single tokens.
- A range of tokens, specified as (*start_token*, *end_token*).
- A comma-delimited list of token ranges — for example, (*start_token1*, *end_token1*) , (*start_token2*, *end_token2*, . . .
- A comma-delimited list of mixed single tokens and token ranges — for example, *token1*, (*start_token2*, *end_token2*) , (*start_token3*, *end_token3*) , *token4*, . . .

--

Separates an option and argument that could be mistaken for a option.

source-dc-name

The name of the datacenter Cassandra uses as the source for streaming. Cassandra rebuilds from any datacenter. If the statement does not specify one, Cassandra chooses at random.

Description

This command operates on multiple nodes in a cluster. Like nodetool bootstrap, `rebuild` only streams data from a single source replica when rebuilding a token range. Use this command to add a new datacenter to an existing cluster.

If `rebuild` fails because some token ranges cannot be retrieved, you can rebuild selectively by using the -ts or --token option to specify a list of tokens, or one or more token ranges.

Note: If `rebuild` is interrupted before completion, you can restart it by re-entering the command. The process resumes from the point at which it was interrupted.

nodetool rebuild_index

Performs a full rebuild of the index for a table

Synopsis

```
$ nodetool <options> rebuild_index -- ( <keyspace> <table> <indexName> ... )
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
keyspace	Name of keyspace.	
table	One or more table names, separated by a space.	
indexName	List of index names separated by a space.	
`--`	Separates an option from an argument that could be mistaken for a option.	

The keyspace and table name followed by a list of index names. For example: `Standard3.IdxName Standard3.IdxName1`

Description

Fully rebuilds one or more indexes for a table.

nodetool refresh

Loads newly placed SSTables onto the system without a restart.

Synopsis

```
$ nodetool <options> refresh -- <keyspace> <table>
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
keyspace	Name of keyspace.	
table	Name of table.	
`--`	Separates an option from an argument that could be mistaken for a option.	

nodetool refreshsizeestimates

Refreshes system.size_estimates table. Use when huge amounts of data are inserted or truncated which can result in size estimates becoming incorrect.

Synopsis

```
$ nodetool <options> refreshsizeestimates
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`--`	Separates an option from an argument that could be mistaken for a option.	

nodetool reloadtriggers

Reloads trigger classes.

Synopsis

```
$ nodetool <options> reloadtriggers
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.

Short	Long	Description
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

nodetool relocatesstables

In Cassandra 3.2 and later, rewrites any SSTable that contains tokens that should be in another data directory for JBOD disks. Basically, this commands relocates SSTables to the correct disk.

Synopsis

```
$ nodetool <options>relocatesstables -- <keyspace> <table>
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
keyspace	Name of keyspace.	
table	Name of table.	
--	Separates an option from an argument that could be mistaken for a option.	

Description

This `nodetool` command can be used to manually rewrite the location of SSTables on disk. It is for use with JBOD disk storage. The command can also be used if you change the replication factor for the cluster stored on JBOD or if you add a new disk. If all the token are correctly stored in the data directories, `nodetool relocatesstables` will have no effect.

Examples

Text

```
$ nodetool relocatesstables cycling
```

nodetool removenode

Provides the status of current node removal, forces completion of pending removal, or removes the identified node.

Synopsis

```
$ nodetool <options> removenode -- <status> | <force> | <ID>
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
status	Status of current node removal.	
force	Forces completion of the pending removal.	
ID	Host ID, in UUID format.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

This command removes a node, shows the status of a removal operation, or forces the completion of a pending removal. When the node is down and `nodetool decommission` cannot be used, use `nodetool removenode`. Run this command only on nodes that are down. If the cluster does not use vnodes, before running the `nodetool removenode` command, adjust the tokens.

Examples

Determine the UUID of the node to remove by running `nodetool status`. Use the UUID of the node that is down to remove the node.

```
$ nodetool status
```

```
Datacenter: DC1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address        Load       Tokens  Owns (effective)  Host ID
                  Rack
UN  192.168.2.101  112.82 KB  256     31.7%             420129fc-0d84-42b0-
be41-ef7dd3a8ad06  RAC1
DN  192.168.2.103  91.11 KB   256     33.9%             d0844a21-3698-4883-
ab66-9e2fd5150edd  RAC1
UN  192.168.2.102  124.42 KB  256     32.6%             8d5ed9f4-7764-4dbd-
bad8-43fddce94b7c  RAC1
```

```
$ nodetool removenode d0844a21-3698-4883-ab66-9e2fd5150edd
```

View the status of the operation to remove the node:

```
$ nodetool removenode status
```

```
RemovalStatus: No token removals in process.
```

Confirm that the node has been removed.

```
$ nodetool removenode status
```

```
Datacenter: DC1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address        Load       Tokens  Owns (effective)  Host ID
                  Rack
UN  192.168.2.101  112.82 KB  256     37.7%             420129fc-0d84-42b0-
be41-ef7dd3a8ad06  RAC1
UN  192.168.2.102  124.42 KB  256     38.3%             8d5ed9f4-7764-4dbd-
bad8-43fddce94b7c  RAC1
```

nodetool repair

Repairs one or more tables.

Synopsis

```
$  nodetool [(-h <host> | --host <host>)] [(-p <port> | --port <port>)]
     [(-pw password | --password password)]
     [(-pwf passwordFilePath | --password-file passwordFilePath)]
     [(-u username | --username username)] repair
     [(-dc specific_dc | --in-dc specific_dc)...]
     [(-dcpar | --dc-parallel)] [(-et end_token | --end-token end_token)]
     [(-full | --full)]
     [(-hosts specific_host | --in-hosts specific_host)...]
     [(-j job_threads | --job-threads job_threads)]
     [(-local | --in-local-dc)] [(-pr | --partitioner-range)]
     [(-seq | --sequential)]
     [(-st start_token | --start-token start_token)] [(-tr | --trace)]
     [--] [keyspace tables...]
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
-dc *specific_dc*	--in-dc *specific_dc*	Repair to nodes in the named datacenter (*specific_dc*).
-dcpar	--dc-parallel	Repair datacenters in parallel, one node per datacenter at a time.
-et *end_token*	--end-token *end_token*	Token UUID. Repair a range of nodes starting with the first token (see *-st*) and ending with this token (*end_token*). Use -hosts to specify neighbor nodes.
-full	--full	Do a full repair.

Short	Long	Description
`-h` *`host_name`*	`--host` *`host_name`*	Node host name or IP address.
`-hosts` *`specific_host`*	`--in-hosts` *`specific_host`*	Repair specific hosts.
`-j` *`job_threads`*	`--job-threads` *`job_threads`*	Number of threads (*job_threads*) to run repair jobs. Usually the number of tables to repair concurrently. Be aware that increasing this setting puts more load on repairing nodes. (Default: 1, maximum: 4)
`-local`	`--in-local-dc`	Use to only repair nodes in the same datacenter.
-pr	--partitioner-range	Run a repair on the partition ranges that are primary on a replica.
`-seq` *`start_token`*	`--sequential` *`start_token`*	Run a sequential repair.
`-st` *`start_token`*	`--start-token` *`start_token`*	Specify the token (*start_token*) at which the repair range starts.
`-tr`	`--trace`	Trace the repair. Traces are logged to *system_traces.events*.
keyspace	Name of keyspace.	
tables	One or more table names, separated by a space.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

Performing an anti-entropy node repair on a regular basis is important, especially in an environment that deletes data frequently. The `repair` command repairs one or more nodes in a cluster, and provides options for restricting repair to a set of nodes. Anti-entropy node repair performs the following tasks:

- Ensures that all data on a replica is consistent.
- Repairs inconsistencies on a node that has been down.

Incremental repair is the default for Cassandra 2.2 and later, and full repair is the default in Cassandra 2.1 and earlier. In Cassandra 2.2 and later, when a full repair is run, SSTables are marked as repaired and anti-compacted. Parallel repair is the default for Cassandra 2.2 and later, and sequential repair is the default in Cassandra 2.1 and earlier.

Using options

You can use options to do these other types of repair:

- Sequential or Parallel
- Full or incremental

Use the -hosts option to list the good nodes to use for repairing the bad nodes. Use -h to name the bad nodes.

Use the -full option for a full repair if required. By default, an incremental repair eliminates the need for constant Merkle tree construction by persisting already repaired data and calculating only the Merkle trees for SSTables that have not been repaired. The repair process is likely more performant than the other types of repair even as datasets grow, assuming you run repairs frequently. Before doing an incremental repair for the first time, perform migration steps first if necessary for tables created before Cassandra 2.2.

Use the -dcpar option to repair data centers in parallel. Unlike sequential repair, parallel repair constructs the Merkle tables for all data centers at the same time. Therefore, no snapshots are required (or generated). Use parallel repair to complete the repair quickly or when you have operational downtime that allows the resources to be completely consumed during the repair.

Performing partitioner range repairs by using the -pr option is generally considered a good choice for doing manual repairs. However, do not use this option with incremental repairs (default for Cassandra 3.0 and later).

Example

All `nodetool repair` arguments are optional.

To do a sequential repair of all keyspaces on the current node:

```
$ nodetool repair -seq
```

To do a partitioner range repair of the bad partition on current node using the good partitions on 10.2.2.20 or 10.2.2.21:

```
$ nodetool repair -pr -hosts 10.2.2.20 10.2.2.21
```

For a start-point-to-end-point repair of all nodes between two nodes on the ring:

```
$ nodetool -st a9fa31c7-f3c0-44d1-b8e7-a26228867840c -et f5bb146c-
db51-475ca44f-9facf2f1ad6e
```

To restrict the repair to the local data center, use the `-dc` option followed by the name of the data center. Issue the command from a node in the data center you want to repair. Issuing the command from a data center other than the named one returns an error. Do not use `-pr` with this option to repair only a local data center.

```
$ nodetool repair -dc DC1
```

Results in output:

```
[2014-07-24 21:59:55,326] Nothing to repair for keyspace 'system'
[2014-07-24 21:59:55,617] Starting repair command #2, repairing 490 ranges
  for keyspace system_traces (seq=true, full=true)
[2014-07-24 22:23:14,299] Repair session 323b9490-137e-11e4-88e3-
c972e09793ca
  for range (820981369067266915,822627736366088177] finished
[2014-07-24 22:23:14,320] Repair session 38496a61-137e-11e4-88e3-
c972e09793ca
  for range (2506042417712465541,2515941262699962473] finished
. . .
```

And an inspection of the system.log shows repair taking place only on IP addresses in DC1.

```
. . .
INFO  [AntiEntropyStage:1] 2014-07-24 22:23:10,708 RepairSession.java:171
  - [repair #16499ef0-1381-11e4-88e3-c972e09793ca] Received merkle tree
  for sessions from /192.168.2.101
INFO  [RepairJobTask:1] 2014-07-24 22:23:10,740 RepairJob.java:145
  - [repair #16499ef0-1381-11e4-88e3-c972e09793ca] requesting merkle trees
  for events (to [/192.168.2.103, /192.168.2.101])
. . .
```

nodetool replaybatchlog

Replay batchlog and wait for finish.

Synopsis

```
$ nodetool <options> replaybatchlog
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Description

`nodetool replaybatchlog` is intended to force a batchlog replay. It also blocks until the batches have been replayed.

nodetool resetlocalschema

Reset the node's local schema and resynchronizes.

Synopsis

```
$ nodetool [options] resetlocalschema [args]
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Note:

- For tarball installations, execute the command from the *install_location*/bin directory.
- If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials.
- `nodetool resetlocalschema` operates on a single node in the cluster if -h is not used to identify one or more other nodes. If the node from which you issue the command is the intended target, you do

not need the -h option to identify the target; otherwise, for remote invocation, identify the target node, or nodes, using -h.

Description

Normally, this command is used to rectify schema disagreements on different nodes. It can be useful if table schema changes have generated too many tombstones, on the order of 100,000s.

`nodetool resetlocalschema` drops the schema information of the local node and resynchronizes the schema from another node. To drop the schema, the tool truncates all the system schema tables. The node will temporarily lose metadata about the tables on the node, but will rewrite the information from another node. If the node is experiencing problems with too many tombstones, the truncation of the tables will eliminate the tombstones.

This command is useful when you have one node that is out of sync with the cluster. The system schema tables must have another node from which to fetch the tables. It is not useful when all or many of your nodes are in an incorrect state. If there is only one node in the cluster (replication factor of 1) – it does not perform the operation, because another node from which to fetch the tables does not exist. Run the command on the node experiencing difficulty.

nodetool resumehandoff

Resume hints delivery process.

Synopsis

```
$ nodetool <options> resumehandoff
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`--`	Separates an option from an argument that could be mistaken for a option.	

nodetool ring

Provides node status and information about the ring.

Synopsis

```
$ nodetool <options> ring ( -r | --resolve-ip ) -- <keyspace>
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.

Short	Long	Description
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`-r`	`--resolve-ip`	Provide node names instead of IP addresses.
keyspace	Name of keyspace.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

Displays node status and information about the ring as determined by the node being queried. This information can give you an idea of the load balance and if any nodes are down. If your cluster is not properly configured, different nodes may show a different ring. Check that the node appears the same way in the ring.If you use virtual nodes (vnodes), use `nodetool status` for succinct output.

- Address

  The node's URL.
- DC (datacenter)

  The datacenter containing the node.
- Rack

  The rack or, in the case of Amazon EC2, the availability zone of the node.
- Status - Up or Down

  Indicates whether the node is functioning or not.
- State - N (normal), L (leaving), J (joining), M (moving)

  The state of the node in relation to the cluster.
- Load - updates every 90 seconds

  The amount of file system data under the cassandra data directory after excluding all content in the snapshots subdirectories. Because all SSTable data files are included, any data that is not cleaned up, such as TTL-expired cell or tombstoned data) is counted.
- Token

  The end of the token range up to and including the value listed. For an explanation of token ranges, see Data Distribution in the Ring .
- Owns

  The percentage of the data owned by the node per datacenter times the replication factor. For example, a node can own 33% of the ring, but show100% if the replication factor is 3.
- Host ID

  The network ID of the node.

nodetool scrub

Rebuild SSTables for one or more Cassandra tables.

Synopsis

```
$ nodetool <options> scrub <keyspace> -- ( -ns | --no-snapshot ) ( -s | --
skip-corrupted ) ( <table> ... )
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
-s	--skip-corrupted	Skip corrupted partitions even when scrubbing counter tables (default false).
-n	--no-validate	Do not validate columns using column validator.
-ns	--no-snapshot	Triggers a snapshot of the scrubbed table first assuming snapshots are not disabled (default).
keyspace	Name of keyspace.	
table	One or more table names, separated by a space.	
--	Separates an option from an argument that could be mistaken for a option.	

Description

Rebuilds SSTables on a node for the named tables and snapshots data files before rebuilding as a safety measure. If possible use nodetool upgradesstables. While scrub rebuilds SSTables, it also discards data that it deems broken and creates a snapshot, which you have to remove manually. If the -ns option is specified, snapshot creation is disabled. If scrub can't validate the column value against the column definition's data type, it logs the partition key and skips to the next partition. Skipping corrupted partitions in tables having counter columns results in under-counting. By default the scrub operation stops if you attempt to skip such a partition. To force the scrub to skip the partition and continue scrubbing, re-run `nodetool scrub` using the --skip-corrupted option.

nodetool setcachecapacity

Set global key and row cache capacities in megabytes.

Synopsis

```
$ nodetool <options> setcachecapacity -- <key-cache-capacity> <row-cache-
capacity>
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.

Short	Long	Description
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
key-cache-capacity	Maximum size in MB of the key cache in memory.	
row-cache-capacity	Maximum size in MB of the row cache in memory.	
counter-cache-capacity	Maximum size in MB of the counter cache in memory.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

The key-cache-capacity argument corresponds to the key_cache_size_in_mb parameter in the cassandra.yaml. Each key cache hit saves one seek and each row cache hit saves a minimum of two seeks. Devoting some memory to the key cache is usually a good tradeoff considering the positive effect on the response time. The default value is empty, which means a minimum of five percent of the heap in MB or 100 MB.

The row-cache-capacity argument corresponds to the row_cache_size_in_mb parameter in the `cassandra.yaml`. By default, row caching is zero (disabled).

The counter-cache-capacity argument corresponds to the counter_cache_size_in_mb in the `cassandra.yaml`. By default, counter caching is a minimum of 2.5% of Heap or 50MB.

nodetool setcachekeystosave

Sets the number of keys saved by each cache for faster post-restart warmup.

Synopsis

```
$ nodetool <options> setcachekeystosave -- <key-cache-keys-to-save> <row-cache-keys-to-save>
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
key-cache-keys-to-save	The number of keys from the key cache to save to the saved caches directory. To disable, set to 0.	

Short	Long	Description
row-cache-keys-to-save	The number of keys from the row cache to save to the saved caches directory. To disable, set to 0.	
counter-cache-keys-to-save	The number of keys from the counter cache to saved to the saved caches directory. To disable, set to 0.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

This command saves the specified number of key and row caches to the saved caches directory, which you specify in the cassandra.yaml. The key-cache-keys-to-save argument corresponds to the key_cache_keys_to_save in the `cassandra.yaml`, which is disabled by default, meaning all keys will be saved. The row-cache-keys-to-save argument corresponds to the row_cache_keys_to_save in the `cassandra.yaml`, which is disabled by default.

nodetool setcompactionthreshold

Sets minimum and maximum compaction thresholds for a table.

Synopsis

```
$ nodetool <options> setcompactionthreshold -- <keyspace> <table>
 <minthreshold> <maxthreshold>
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
keyspace	Name of a keyspace.	
table	Name of a table.	
minthreshold	Sets the minimum number of SSTables to trigger a minor compaction when using SizeTieredCompactionStrategy or DateTieredCompactionStrategy.	
maxthreshold	Sets the maximum number of SSTables to allow in a minor compaction when using SizeTieredCompactionStrategy or DateTieredCompactionStrategy.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

This parameter controls how many SSTables of a similar size must be present before a minor compaction is scheduled. The max_threshold table property sets an upper bound on the number of SSTables that may be compacted in a single minor compaction, as described in http://wiki.apache.org/cassandra/MemtableSSTable.

When using LeveledCompactionStrategy, maxthreshold sets the MAX_COMPACTING_L0, which limits the number of L0 SSTables that are compacted concurrently to avoid wasting memory or running out of memory when compacting highly overlapping SSTables.

nodetool setcompactionthroughput

Sets the throughput capacity for compaction in the system, or disables throttling.

Synopsis

```
$ nodetool <options> setcompactionthroughput -- <value_in_mb>
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
value_in_mb	The throughput capacity in MB per second for compaction. To disable throttling, set to 0.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

Set value_in_mb to 0 to disable throttling.

nodetool sethintedhandoffthrottlekb

Sets hinted handoff throttle in kb/sec per delivery thread.

Synopsis

```
$ nodetool <options> sethintedhandoffthrottlekb <value_in_kb/sec>
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
value_in_kb/sec	Throttle time in kilobytes per second.	

Short	Long	Description
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

When a node detects that a node for which it is holding hints has recovered, it begins sending the hints to that node. This setting specifies the maximum sleep interval per delivery thread in kilobytes per second after delivering each hint. The interval shrinks proportionally to the number of nodes in the cluster. For example, if there are two nodes in the cluster, each delivery thread uses the maximum interval; if there are three nodes, each node throttles to half of the maximum interval, because the two nodes are expected to deliver hints simultaneously.

Example

```
$ nodetool sethintedhandoffthrottlekb 2048
```

nodetool setlogginglevel

Set the log level for a service.

Synopsis

```
$ nodetool <options> setlogginglevel -- <class> <level>
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
class	The class for changing the level, a fully qualified domain name such as org.apache.cassandra.service.StorageProxy.	
level	Logging level, for example DEBUG.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

You can use this command to set logging levels for services instead of modifying the logback-text.xml file. The following values are valid for the logger class qualifier:

- org.apache.cassandra
- org.apache.cassandra.db
- org.apache.cassandra.service.StorageProxy

The possible log levels are:

- ALL
- TRACE

- DEBUG
- INFO
- WARN
- ERROR
- OFF

If both class qualifier and level arguments to the command are empty or null, the command resets logging to the initial configuration.

Example

This command sets the StorageProxy service to debug level.

```
$ nodetool setlogginglevel org.apache.cassandra.service.StorageProxy DEBUG
```

Note: Cassandra 3.0 and later support extended logging for Compaction. This utility must be configured as part of the table configuration. The extended compaction logs are stored in a separate file. For details, see Enabling extended compaction logging.

nodetool setstreamthroughput

Sets the throughput capacity in Mb (megabits) for streaming in the system, or disables throttling.

Synopsis

```
$ nodetool <options> setstreamthroughput -- <value_in_mb>
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
value_in_mb	Throughput capacity in megabits per second for streaming. To disable, set to 0.	
--	Separates an option from an argument that could be mistaken for a option.	

Description

Set value_in_mb to 0 to disable throttling.

nodetool settimeout

Set the specified timeout in milliseconds, or 0 to disable timeout. (Cassandra 3.4 and later).

Synopsis

```
$ nodetool [options] settimeout [--] <timeout_type> <timeout_in_ms>
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
timeout_type	Type of timeout. Type should be one of read, range, write, counterwrite, cascontention, truncate, streamingsocket, misc (general rpc_timeout_in_ms).	
timeout_in_ms	Timeout in in milliseconds. To disable socket streaming, set to 0.	
--	Separates an option from an argument that could be mistaken for a option.	

Note:

- For tarball installations, execute the command from the *install_location*/bin directory.
- If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials.
- -- separates an option and argument that could be mistaken for a option.
- The timeout type:
  - read
  - range
  - write
  - counterwrite
  - cascontention
  - truncate
  - streamingsocket
  - misc, such as general rpc_timeout_in_ms

Description

The `nodetool gettimeout` command sets the specified timeout in milliseconds. Use "0" to disable a timeout. Several timeouts are available.

Examples

```
$ nodetool -u cassandra -pw cassandra settimeout read 100
```

nodetool settraceprobability

Sets the probability for tracing a request.

Synopsis

```
$ nodetool <options> settraceprobability -- <value>
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
value	Trace probability between 0 and 1. For example: 0.2.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

Probabilistic tracing is useful to determine the cause of intermittent query performance problems by identifying which queries are responsible. This option traces some or all statements sent to a cluster. Tracing a request usually requires at least 10 rows to be inserted.

A probability of 1.0 will trace everything whereas lesser amounts (for example, 0.10) only sample a certain percentage of statements. Care should be taken on large and active systems, as system-wide tracing will have a performance impact. Unless you are under very light load, tracing all requests (probability 1.0) will probably overwhelm your system. Start with a small fraction, for example, 0.001 and increase only if necessary. The trace information is stored in a system_traces keyspace that holds two tables – sessions and events, which can be easily queried to answer questions, such as what the most time-consuming query has been since a trace was started. Query the parameters map and thread column in the system_traces.sessions and events tables for probabilistic tracing information.

To discover the current trace probability setting, use nodetool gettraceprobability.

nodetool snapshot

Take a snapshot of one or more keyspaces, or of a table, to backup data.

Synopsis

```
$ nodetool <options> snapshot
     ( -cf <table> | --column-family <table> )
     (-kc <ktlist> | --kc.list <ktlist> | -kt <ktlist> | --kt-list <ktlist>)
     (-sf | --skip-flush)
     (-t <tag> | --tag <tag> )
     -- ( <keyspace> ) | ( <keyspace> ... )
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.

Short	Long	Description
`-u`	`--username`	Remote JMX agent username.
`-cf` *`table`*	`--column-family` *`table`*	Name of the *table* to snapshot. You must specify one and only one keyspace.
	`--table` *`table`*	Name of the *table* to snapshot. You must specify one and only one keyspace.
`-kc` *`ktlist`*	`--kc.list` *`ktlist`*	The list of keyspace.tables to snapshot. Requires list of keyspaces.
`-kt` *`ktlist`*	`--kt-list` *ktlist*	The list of keyspace.tablea to snapshot. Requires list of keyspaces.
`-sf`	`--skip-flush`	Executes the snapshot without flushing the tables first (Cassandra 3.4 and later).
`-t`	`--tag`	Name of snapshot.
keyspace	One or more optional keyspace names, separated by a space. Default: all keyspaces	
`--`		Separates an option from an argument that could be mistaken for a option.

Description

Use this command to back up data using a snapshot. See the examples below for various options.

Cassandra flushes the node before taking a snapshot, takes the snapshot, and stores the data in the snapshots directoryof each keyspace in the data directory. If you do not specify the name of a snapshot directory using the -t option, Cassandra names the directory using the timestamp of the snapshot, for example 1391460334889. Follow the procedure for taking a snapshot before upgrading Cassandra. When upgrading, backup all keyspaces. For more information about snapshots, see Apache documentation.

Example: All keyspaces

Take a snapshot of all keyspaces on the node. On Linux, in the Cassandra `bin` directory, for example:

```
$ nodetool snapshot
```

The following message appears:

```
Requested creating snapshot(s) for [all keyspaces] with snapshot name
 [1391464041163]
Snapshot directory: 1391464041163
```

Because you did not specify a snapshot name, Cassandra names snapshot directories using the timestamp of the snapshot. If the keyspace contains no data, empty directories are not created.

Example: Single keyspace snapshot

Assuming you created the keyspace cycling, took a snapshot of the keyspace and named the snapshot 2015.07.17.:

```
$ nodetool snapshot -t 2015.07.17 cycling
```

The following output appears:

```
Requested creating snapshot(s) for [cycling] with snapshot name [2015.07.17]
```

```
Snapshot directory: 2015.07.17
```

Assuming the cycling keyspace contains two tables, cyclist_name and upcoming_calendar, taking a snapshot of the keyspace creates multiple snapshot directories named 2015.07.17. A number of .db files containing the data are located in these directories. For example, from the installation directory:

```
$ cd data/data/cycling/cyclist_name-a882dca02aaf11e58c7b8b496c707234/
snapshots/2015.07.17
$ ls
```

```
la-1-big-CompressionInfo.db  la-1-big-Index.db       la-1-big-TOC.txt
la-1-big-Data.db             la-1-big-Statistics.db  la-1-big-Digest.adler32
la-1-big-Filter.db           la-1-big-Summary.db  manifest.json
```

```
$ cd data/data/cycling/cyclist_name-a882dca02aaf11e58c7b8b496c707234/
snapshots/2015.07.17
$ ls
```

```
la-1-big-CompressionInfo.db  la-1-big-Index.db       la-1-big-TOC.txt
la-1-big-Data.db             la-1-big-Statistics.db  la-1-big-Digest.adler32
la-1-big-Filter.db           la-1-big-Summary.db  manifest.json
```

Example: Multiple keyspaces snapshot

Assuming you created a keyspace named mykeyspace in addition to the cycling keyspace, take a snapshot of both keyspaces.

```
$ nodetool snapshot mykeyspace cycling
```

The following message appears:

```
Requested creating snapshot(s) for [mykeyspace, cycling] with snapshot name
 [1391460334889]
Snapshot directory: 1391460334889
```

Example: Single table snapshot

Take a snapshot of only the cyclist_name table in the cycling keyspace.

```
$ nodetool snapshot --table cyclist_name cycling
```

```
Requested creating snapshot(s) for [cycling] with snapshot name
 [1391461910600]
Snapshot directory: 1391461910600
```

Cassandra creates the snapshot directory named 1391461910600 that contains the backup data of cyclist_name table in `data/data/cycling/cyclist_name-a882dca02aaf11e58c7b8b496c707234/snapshots`, for example.

Example: List of different keyspace.tables snapshot

Take a snapshot of several tables in different keyspaces, such as the cyclist_name table in the cycling keyspace and the sample_times table in the test keyspace. The keyspace.table list should be comma-delimited with no spaces.

```
$ nodetool snapshot -kt cycling.cyclist_name,test.sample_times
```

```
Requested creating snapshot(s) for [cycling.cyclist_name,test.sample_times]
 with snapshot name [1431045288401]
Snapshot directory: 1431045288401
```

nodetool status

Provide information about the cluster, such as the state, load, and IDs.

Synopsis

```
$ nodetool <options> status ( -r | --resolve-ip ) -- <keyspace>
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`-r`	`--resolve-ip`	Show node names instead of IP addresses.
keyspace	Name of keyspace.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

The status command provides the following information:

- Status - U (up) or D (down)

  Indicates whether the node is functioning or not.
- State - N (normal), L (leaving), J (joining), M (moving)

  The state of the node in relation to the cluster.
- Address

  The node's URL.
- Load - updates every 90 seconds

  The amount of file system data under the cassandra data directory after excluding all content in the snapshots subdirectories. Because all SSTable data files are included, any data that is not cleaned up, such as TTL-expired cell or tombstoned data) is counted.
- Tokens

  The number of tokens set for the node.
- Owns

  The percentage of the data owned by the node per datacenter times the replication factor. For example, a node can own 33% of the ring, but show 100% if the replication factor is 3.

Attention: If your cluster uses keyspaces having different replication strategies or replication factors, specify a keyspace when you run `nodetool status` to get meaningful ownership information.

- Host ID

  The network ID of the node.
- Rack

  The rack or, in the case of Amazon EC2, the availability zone of the node.

Example

This example shows the output from running nodetool status.

```
$ nodetool status mykeyspace

Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Tokens  Owns    Host ID
  Rack
UN  127.0.0.1  47.66 KB   1       33.3%   aaa1b7c1-6049-4a08-ad3e-3697a0e30e10
  rack1
UN  127.0.0.2  47.67 KB   1       33.3%   1848c369-4306-4874-afdf-5c1e95b8732e
  rack1
UN  127.0.0.3  47.67 KB   1       33.3%   49578bf1-728f-438d-b1c1-d8dd644b6f7f
  rack1
```

nodetool statusbackup

Synopsis

```
$ nodetool <options> statusbackup
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

Provides the status of backup.

nodetool statusbinary

Provide the status of native transport.

Synopsis

```
$ nodetool <options> statusbinary
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Description

Provides the status of the binary protocol, also known as the native transport.

nodetool statusgossip

Synopsis

```
$ nodetool <options> statusgossip
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Description

Provides the status of gossip.

nodetool statushandoff

Synopsis

```
$ nodetool <options> statushandoff
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Description

Provides the status of hinted handoff.

nodetool statusthrift

Provide the status of the Thrift server.

Synopsis

```
$ nodetool <options> statusthrift
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

nodetool stop

Stops the compaction process.

Synopsis

```
$ nodetool <options> stop -- <compaction_type>
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.

Short	Long	Description
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
compaction type	Supported types are COMPACTION, VALIDATION, CLEANUP, SCRUB, VERIFY, INDEX_BUILD.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Note:

- For tarball installations, execute the command from the *install_location*/bin directory.
- If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials.
- `nodetool stop` operates on a single node in the cluster if `-h` is not used to identify one or more other nodes. If the node from which you issue the command is the intended target, you do not need the `-h` option to identify the target; otherwise, for remote invocation, identify the target node, or nodes, using `-h`.
- Valid compaction types: COMPACTION, VALIDATION, CLEANUP, SCRUB, INDEX_BUILD

Description

Stops all compaction operations from continuing to run. This command is typically used to stop a compaction that has a negative impact on the performance of a node. After the compaction stops, Cassandra continues with the remaining operations in the queue. Eventually, Cassandra restarts the compaction.

In Cassandra 2.2 and later, a single compaction operation can be stopped with the `-id` option. Run `nodetool compactionstats` to find the compaction ID.

nodetool stopdaemon

Stops the cassandra daemon.

Synopsis

```
$ nodetool <options> stopdaemon
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`--`	Separates an option from an argument that could be mistaken for a option.	

nodetool tablehistograms

Provides statistics about a table that could be used to plot a frequency function.

Synopsis

```
$ nodetool <options> tablehistograms -- <keyspace>.<table>
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
keyspace	Name of keyspace.	
table	One or more table names, separated by a space. **Note:** Either *keyspace.table* or *keyspace table* can be used to designate the table.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

The `nodetool tablehistograms` command provides statistics about a table, including read/write latency, partition size, column count, and number of SSTables. The report is incremental, not cumulative. It covers all operations since the last time `nodetool tablehistograms` was run in the current session. The use of the metrics-core library makes the output more informative and easier to understand.

Example

For example, to get statistics about the libout table in the libdata keyspace, use this command:

```
$ %CASSANDRA_HOME%/bin/nodetool tablehistograms libdata libout
```

Output:

```
libdata/libout histograms
Percentile  SSTables     Write Latency      Read Latency    Partition Size
      Cell Count
                              (micros)          (micros)           (bytes)

50%             0.00             39.50             36.00              1597
              42
75%             0.00             49.00             55.00              1597
              42
95%             0.00             95.00             82.00              8239
             258
98%             0.00            126.84            110.42             17084
             446
99%             0.00            155.13            123.71             24601
             770
```

```
Min             0.00              3.00              3.00              1110
              36
Max             0.00          50772.00            314.00            126934
            3973
```

The output shows the percentile rank of read and write latency values, the partition size, and the cell count for the table.

nodetool tablestats

Provides statistics about one or more tables.

Synopsis

```
$ nodetool [ options ] tablestats
[ -H | --human-readable ]
[ -i  table [, table ] . . . ] [ - - ]
[ keyspace | table | keyspace.table ] [keyspace | table | keyspace.table
 ] . . .
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`-F` *format*	`--format` *format*	Output format: `json` or `yaml`.
`-H`	`--human-readable`	Display bytes in human readable form: KiB (kibibyte), MiB (mebibyte), GiB (gibibyte), TiB (tebibyte).
`-i`		Ignore the list of tables and display the remaining tables.
keyspace.table	List of tables (or keyspace) names.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

The `nodetool tablestats` command provides statistics about one or more tables. It's updated when SSTables change through compaction or flushing. Cassandra uses the metrics-core library to make the output more informative and easier to understand.

Table: nodetool tablestats output for a single table

Name of statistic	Example value	Brief description	Related information
Keyspace	libdata	Name of the keyspace	Keyspace and table
Table	libout	Name of this table	

Name of statistic	Example value	Brief description	Related information
SSTable count	3	Number of SSTables containing data for this table	How to use the SSTable counts metric
Space used (live)	9592399	Total number of bytes of disk space used by all active SSTables belonging to this table	Storing data on disk in SSTables
Space used (total)	9592399	Total number of bytes of disk space used by SSTables belonging to this table, including obsolete SSTables waiting to be GCd	Same as above.
Space used by snapshots (total):	0	Total number of bytes of disk space used by snapshot of this table's data	About snapshots on page 152
Off heap memory used (total)		Total number of bytes of off heap memory used for memtables, Bloom filters, index summaries and compression metadata for this table	
SSTable Compression Ratio	0.367…	Ratio of size of compressed SSTable data to its uncompressed size	Types of compression options.
Number of keys (estimate)	3	The number of partition keys for this table	Not the number of primary keys. This gives you the estimated number of partitions in the table.
Memtable cell count	1022550	Number of cells (storage engine rows x columns) of data in the memtable for this table	Cassandra memtable structure in memory
Memtable data size	32028148	Total number of bytes in the memtable for this table	Total amount of live data stored in the memtable, excluding any data structure overhead.
Memtable off heap memory used	0	Total number of bytes of off-heap data for this memtable, including column related overhead and partitions overwritten	The maximum amount is set in cassandra.yaml by the property memtable_offheap_space_in_mb.
Memtable switch count	3	Number of times a full memtable for this table was swapped for an empty one	Increases each time the memtable for a table is flushed to disk. See How memtables are measured article.
Local read count	11207	Number of requests to read tables in the keyspace since startup	

Name of statistic	Example value	Brief description	Related information
Local read latency	0.048 ms	Round trip time in milliseconds to complete the most recent request to read the table	Factors that affect read latency
Local write count	17598	Number of local requests to update the table since startup	
Local write latency	0.054 ms	Round trip time in milliseconds to complete an update to the table	Factors that affect write latency
Pending flushes	0	Estimated number of reads, writes, and cluster operations pending for this table	**Important:** Monitor this metric to watch for blocked or overloaded memtable flush writers. The nodetool tpstats tool does not report on blocked flushwriters.
Bloom filter false positives	0	Number of false positives reported by this table's Bloom filter	Tuning bloom filters
Bloom filter false ratio	0.00000	Fraction of all bloom filter checks resulting in a false positive from the most recent read	
Bloom filter space used, bytes	11688	Size in bytes of the bloom filter data for this table	
Bloom filter off heap memory used	8	The number of bytes of off heap memory used for Bloom filters for this table	
Index summary off heap memory used	41	The number of bytes of off heap memory used for index summaries for this table	
Compression metadata off heap memory used	8	The number of bytes of off heap memory used for compression offset maps for this table	
Compacted partition minimum	1110	Size in bytes of the smallest compacted partition for this table	
Compacted partition maximum bytes	126934	Size in bytes of the largest compacted partition for this table	
Compacted partition mean bytes	2730	The average size of compacted partitions for this table	
Average live cells per slice (last five minutes)	0.0	Average number of cells scanned by single key queries during the last five minutes	

Name of statistic	Example value	Brief description	Related information
Maximum live cells per slice (last five minutes)	0.0	Maximum number of cells scanned by single key queries during the last five minutes	
Average tombstones per slice (last five minutes)	0.0	Average number of tombstones scanned by single key queries during the last five minutes	
Maximum tombstones per slice (last five minutes)	0.0	Maximum number of tombstones scanned by single key queries during the last five minutes	
Dropped mutations	0.0	The number of mutations (INSERTs, UPDATEs or DELETEs) started on this table but not completed	A high number of dropped mutations can indicate an overloaded node.

Examples

An excerpt of the output of the command reporting on a library data table just flushed to disk.

```
$ nodetool tablestats keyspace1.standard1
Keyspace: keyspace1
 Read Count: 182849
 Read Latency: 0.11363755339104945 ms.
 Write Count: 435355
 Write Latency: 0.01956930550929701 ms.
 Pending Flushes: 0
  Table: standard1
  SSTable count: 2
  Space used (live): 54131487
  Space used (total): 54131487
  Space used by snapshots (total): 0
  Off heap memory used (total): 309620
  SSTable Compression Ratio: 0.0
  Number of keys (estimate): 376390
  Memtable cell count: 200120
  Memtable data size: 47355786
  Memtable off heap memory used: 0
  Memtable switch count: 2
  Local read count: 182849
  Local read latency: 0.125 ms
  Local write count: 435355
  Local write latency: 0.022 ms
  Pending flushes: 0
  Bloom filter false positives: 11
  Bloom filter false ratio: 0.00009
  Bloom filter space used: 272192
  Bloom filter off heap memory used: 272176
  Index summary off heap memory used: 37444
  Compression metadata off heap memory used: 0
  Compacted partition minimum bytes: 216
  Compacted partition maximum bytes: 258
  Compacted partition mean bytes: 258
  Average live cells per slice (last five minutes): 1.0
  Maximum live cells per slice (last five minutes): 1
  Average tombstones per slice (last five minutes): 1.0
```

```
   Maximum tombstones per slice (last five minutes): 1
```

Using the human-readable option

Use the human-readable `-H` option to get output in easier-to-read units. For example:

```
$ C:\> %CASSANDRA_HOME%nodetool tablestats -H keyspace1.standard1
Keyspace: keyspace1
 Read Count: 182849
 Read Latency: 0.11363755339104945 ms.
 Write Count: 435355
 Write Latency: 0.01956930550929701 ms.
 Pending Flushes: 0
  Table: standard1
  SSTable count: 2
  Space used (live): 51.62 MB
  Space used (total): 51.62 MB
  Space used by snapshots (total): 0 bytes
  Off heap memory used (total): 302.36 KB
  SSTable Compression Ratio: 0.0
  Number of keys (estimate): 376390
  Memtable cell count: 200120
  Memtable data size: 45.16 MB
  Memtable off heap memory used: 0 bytes
  Memtable switch count: 2
  Local read count: 182849
  Local read latency: 0.125 ms
  Local write count: 435355
  Local write latency: 0.022 ms
  Pending flushes: 0
  Bloom filter false positives: 11
  Bloom filter false ratio: 0.00000
  Bloom filter space used: 265.81 KB
  Bloom filter off heap memory used: 265.8 KB
  Index summary off heap memory used: 36.57 KB
  Compression metadata off heap memory used: 0 bytes
  Compacted partition minimum bytes: 216 bytes
  Compacted partition maximum bytes: 258 bytes
  Compacted partition mean bytes: 258 bytes
  Average live cells per slice (last five minutes): 1.0
  Maximum live cells per slice (last five minutes): 1
  Average tombstones per slice (last five minutes): 1.0
  Maximum tombstones per slice (last five minutes): 1
```

nodetool toppartitions

Samples database reads and writes and reports the most active partitions in a specified table.

Synopsis

```
$ nodetool [ options ] toppartitions
[ -a samplers ] [ -k topcount ] [ -s size ] [ -- ]
keyspace table duration
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.

Short	Long	Description
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`-a` *`samplers`*		Comma separated list of samplers to use (default: all)
`-k` *`topCount`*		The number of the top partitions to list (default: 10)
`-s` *`size`*		The capacity of stream summary. A value closer to the actual cardinality of partitions yields more accurate results. (default: 256)
keyspace	Name of keyspace.	
table	Name of table.	
duration	The duration in milliseconds	
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

The `nodetool toppartitions` command samples the activity in a table during the specified duration and prints lists of the most active partitions during that time period. To run this command you must specify the keyspace and table to focus on and the time interval (in milliseconds) during which Cassandra samples the table's activity.

Examples

Sample the most active partitions for the table `test.users` for 1,000 milliseconds

```
nodetool toppartitions test users 1000
```

The output of `nodetool toppartitions` is similar to the following:

```
WRITES Sampler:
  Cardinality: ~2 (256 capacity)
  Top 4 partitions:
 Partition                Count       +/-
 4b504d39354f37353131        15        14
 3738313134394d353530        15        14
 4f363735324e324e4d30        15        14
 303535324e4b4d504c30        15        14

READS Sampler:
  Cardinality: ~3 (256 capacity)
  Top 4 partitions:
 Partition                Count       +/-
       4d4e30314f374e313730        42        41
 4f363735324e324e4d30        42        41
 303535324e4b4d504c30        42        41
 4e355030324e344d3030        41        40
```

For each of the samplers used (`WRITES` and `READS` in the example), `toppartitions` reports:

- The cardinality of the sampled operations (that is, the number of unique operations in the sample set)

- The n partitions in the specified table that had the most traffic in the specified time period (where n is the value of the -k argument, or ten if -k is not explicitly set in the command).

  For each Partition, toppartitions reports:

Partition

The partition key

Count

The number of operations of the specified type that occurred during the specified time period.

+/-

The margin of error for the **Count** statistic

Note: To keep the toppartitions reporting from slowing performance, Cassandra does not keep an exact count of operations, but uses sampling techniques to create an approximate number. (This example reports on a sample cluster; a production system might generate millions of reads or writes in a few seconds.) The +/- figure allows you to judge the accuracy of the toppartitions reporting.

nodetool tpstats

Provides usage statistics of thread pools.

Synopsis

```
$ nodetool <options> tpstats
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

Description

Cassandra is based on a Staged Event Driven Architecture (SEDA). Cassandra separates different tasks into stages connected by a messaging service. Each stage has a queue and a thread pool. Although some stages skip the messaging service and queue tasks immediately on a different stage when it exists on the same node. Cassandra can back up a queue if the next stage is too busy and lead to a performance bottlenecks, as described in http://wiki.apache.org/cassandra/Operations#Monitoring.

The nodetool tpstats command reports on each stage of Cassandra operations by thread pool:

- The number of Active threads
- The number of Pending requests waiting to be executed by this thread pool
- The number of tasks Completed by this thread pool
- The number of requests that are currently Blocked because the thread pool for the next step in the service is full
- The total number of All-Time Blocked requests, which are all requests blocked in this thread pool up to now.

Reports are updated when SSTables change through compaction or flushing.

Run `nodetool tpstats` on a local node to get statistics for the thread pool used by the Cassandra instance running on that node.

Run `nodetool tpstats` with the appropriate options to check the thread pool statistics for a remote node. For setup instructions, see Secure JMX Authentication.

nodetool tpstats pool names and tasks

This table describes the Cassandra task or property associated with each pool name reported in the `nodetool tpstats` output:

Pool Name	Associated tasks	Related information
AntiEntropyStage	Processing repair messages and streaming	For details, see Nodetool repair.
CacheCleanupExecutor	Clearing the cache	
CommitlogArchiver	Copying or archiving commitlog files for recovery	
CompactionExecutor	Running compaction	
CounterMutationStage	Processing local counter changes	Will back up if the write rate exceeds the mutation rate. A high pending count will be seen if consistency level is set to ONE and there is a high counter increment workload.
GossipStage	Distributing node information via Gossip	Out of sync schemas can cause issues. You may have to sync using `nodetool resetlocalschema` .
HintedHandoff	Sending missed mutations to other nodes	Usually symptom of a problem elsewhere. Use `nodetool disablehandoff` and run repair.
InternalResponseStage	Responding to non-client initiated messages, including bootstrapping and schema checking	
MemtableFlushWriter	Writing memtable contents to disk	May back up if the queue is overruns the disk I/O, or because of sorting processes. **Warning:** `nodetool tpstats` no longer reports blocked threads in the MemtableFlushWriter pool. Check the Pending Flushes metric reported by `nodetool tblestats`.
MemtablePostFlush	Cleaning up after after flushing the memtable (discarding commit logs and secondary indexes as needed)	
MemtableReclaimMemory	Making unused memory available	
MigrationStage	Processing schema changes	
MiscStage	Snapshotting, replicating data after node remove completed.	

Pool Name	Associated tasks	Related information
MutationStage	Performing local inserts/updates, schema merges, commit log replays or hints in progress	A high number of `Pending` write requests indicates the node is having a problem handling them. Fix this by adding a node, tuning hardware and configuration, and/or updating data models.
Native-Transport-Requests	Processing CQL requests to the server	
PendingRangeCalculator	Calculating pending ranges per bootstraps and departed nodes	Reporting by this tool is not useful — see Developer notes
ReadRepairStage	Performing read repairs	Usually fast, if there is good connectivity between replicas. If `Pending` grows too large, attempt to lower the rate for high-read tables by altering the table to use a smaller read_repair_chance value, like 0.11.
ReadStage	Performing local reads	Also includes deserializing data from row cache. Pending values can cause increased read latency. Generally resolved by adding nodes or tuning the system.
RequestResponseStage	Handling responses from other nodes	
ValidationExecutor	Validating schema	

nodetool tpstats droppable messages

Cassandra generates the messages listed below, but discards them after a timeout. The `nodetool tpstats` command reports the number of messages of each type that have been dropped. You can view the messages themselves using a JMX client.

Message Type	Stage	Notes
BINARY	n/a	Deprecated
_TRACE	n/a (special)	Used for recording traces (nodetool settraceprobability) Has a special executor (1 thread, 1000 queue depth) that throws away messages on insertion instead of within the execute
MUTATION	MutationStage	If a write message is processed after its timeout (write_request_timeout_in_ms) it either sent a failure to the client or it met its requested consistency level and will relay on hinted handoff and read repairs to do the mutation if it succeeded.
COUNTER_MUTATION	MutationStage	If a write message is processed after its timeout (write_request_timeout_in_ms) it

Message Type	Stage	Notes
		either sent a failure to the client or it met its requested consistency level and will relay on hinted handoff and read repairs to do the mutation if it succeeded.
READ_REPAIR	MutationStage	Times out after write_request_timeout_in_ms
READ	ReadStage	Times out after read_request_timeout_in_ms. No point in servicing reads after that point since it would of returned error to client
RANGE_SLICE	ReadStage	Times out after range_request_timeout_in_ms.
PAGED_RANGE	ReadStage	Times out after request_timeout_in_ms.
REQUEST_RESPONSE	RequestResponseStage	Times out after request_timeout_in_ms. Response was completed and sent back but not before the timeout

Example

Running `nodetool tpstats` on the host `labcluster`:

```
$ nodetool -h labcluster tpstats
```

Example output is:

```
Pool Name                    Active   Pending      Completed   Blocked  All
 time blocked
CounterMutationStage              0         0              0         0
           0
ReadStage                         0         0            103         0
           0
RequestResponseStage              0         0              0         0
           0
MutationStage                     0         0       13234794         0
           0
ReadRepairStage                   0         0              0         0
           0
GossipStage                       0         0              0         0
           0
CacheCleanupExecutor              0         0              0         0
           0
AntiEntropyStage                  0         0              0         0
           0
MigrationStage                    0         0             11         0
           0
ValidationExecutor                0         0              0         0
           0
CommitLogArchiver                 0         0              0         0
           0
```

```
MiscStage                          0         0              0         0
            0
MemtableFlushWriter                0         0            126         0
            0
MemtableReclaimMemory              0         0            126         0
            0
PendingRangeCalculator             0         0              1         0
            0
MemtablePostFlush                  0         0           1468         0
            0
CompactionExecutor                 0         0            254         0
            0
InternalResponseStage              0         0              1         0
            0
HintedHandoff                      0         0              0

Message type           Dropped
RANGE_SLICE                  0
READ_REPAIR                  0
PAGED_RANGE                  0
BINARY                       0
READ                         0
MUTATION                   180
_TRACE                       0
REQUEST_RESPONSE             0
COUNTER_MUTATION             0
```

nodetool truncatehints

Truncates all hints on the local node, or truncates hints for the one or more endpoints.

Synopsis

```
$ nodetool <options> truncatehints -- ( <endpoint> ... )
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
endpoint	One or more endpoint IP addresses or host names designating which hints to deleted.	
--	Separates an option from an argument that could be mistaken for a option.	

nodetool upgradesstables

Rewrites SSTables for tables that are not running the current version of Cassandra.

Synopsis

```
$ nodetool <options> upgradesstables
 ( -a | --include-all-sstables )
 -- <keyspace> ( <table> ... )
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
-a	--include-all-sstables	Snapshot name.
keyspace	Name of keyspace.	
table	One or more table names, separated by a space.	
--	Separates an option from an argument that could be mistaken for a option.	

Description

Rewrites SSTables on a node that are incompatible with the current version. Use this command when upgrading your server or changing compression options.

nodetool viewbuildstatus

Shows the progress of a materialized view build.

Synopsis

```
$ nodetool viewbuildstatus <keyspace> <view> | <keyspace.view>
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
keyspace	The name of the keyspace.	
view	The name of the view. You can also use *keyspace.view*.	

Short	Long	Description
`--`	Separates an option from an argument that could be mistaken for a option.	

Description

Shows the progress of a materialized view build.

nodetool verify

Verify (check data checksum for) one or more tables.

Synopsis

```
$ nodetool [options] verify [(-e | --extended-verify)] [--] [<keyspace>
 <tables>...]
```

Table: Options

Short	Long	Description
`-h`	`--host`	Hostname or IP address.
`-p`	`--port`	Port number.
`-pwf`	`--password-file`	Password file path.
`-pw`	`--password`	Password.
`-u`	`--username`	Remote JMX agent username.
`-e`	`--extended-verify`	Verify each cell data, beyond simply checking SSTable checksums.
`keyspace`	Name of keyspace.	
table	One or more table names, separated by a space.	
`--`	Separates an option from an argument that could be mistaken for a option.	

Note:

- For tarball installations, execute the command from the *install_location*/bin directory.
- If a username and password for RMI authentication are set explicitly in the cassandra-env.sh file for the host, then you must specify credentials.
- `nodetool verify` operates on a single node in the cluster if -h is not used to identify one or more other nodes. If the node from which you issue the command is the intended target, you do not need the -h option to identify the target; otherwise, for remote invocation, identify the target node, or nodes, using -h.

Description

The `nodetool verify` command checks the data checksum for one or more specified tables. An optional argument, `-e` or `--extended-verify`, will verify each cell data, whereas without the option, only the SSTable checksums are verified.

Examples

```
$ nodetool -u cassandra -pw cassandra verify cycling cyclist_name
```

nodetool version

Provides the version number of Cassandra running on the specified node.

Synopsis

```
$ nodetool <options> version
```

Table: Options

Short	Long	Description
-h	--host	Hostname or IP address.
-p	--port	Port number.
-pwf	--password-file	Password file path.
-pw	--password	Password.
-u	--username	Remote JMX agent username.
--	Separates an option from an argument that could be mistaken for a option.	

The cassandra utility

You can run Cassandra 3.2 and later with start-up parameters to by adding them to the `jvm.options` file (package or tarball installations). You can also enter parameters at the command line when starting up a tarball installation.

Note: If you are using Cassandra 3.1, see the Cassandra 3.0 documentation.

You can also add options such as maximum and minimum heap size to the `jvm.options` file to pass them to the Java virtual machine at startup, rather than setting them in the environment.

Usage

Add a parameter to the jvm.options file as follows:

```
JVM_OPTS="$JVM_OPTS -D[PARAMETER]"
```

When starting up a tarball installations, you can add parameters at the command line:

```
$ cassandra [PARAMETERS]
```

Examples:

- Command line: `$ bin/cassandra -Dcassandra.load_ring_state=false`
- jvm.options: `JVM_OPTS="$JVM_OPTS -Dcassandra.load_ring_state=false"`

The Example section contains more examples.

Command line only options

Option	Description
-f	Start the cassandra process in foreground. The default is to start as background process.
-h	Help.
-p *filename*	Log the process ID in the named file. Useful for stopping Cassandra by killing its PID.
-v	Print the version and exit.

Start-up parameters

The -D option specifies start-up parameters at the command line and in the cassandra-env.sh file.

cassandra.auto_bootstrap=false

Sets auto_bootstrap to `false` on initial set-up of the cluster. The next time you start the cluster, you do not need to change the cassandra.yaml file on each node to revert to `true`.

cassandra.available_processors=*number_of_processors*

In a multi-instance deployment, each Cassandra instance independently assumes that all CPU processors are available to it. Use this setting to specify a smaller set of processors.

cassandra.boot_without_jna=true

Configures Cassandra to boot without JNA. If you do not set this parameter to `true`, and JNA does not initalize, Cassandra does not boot.

cassandra.config=*directory*

Sets the directory location of the cassandra.yaml file. The default location depends on the type of installation.

cassandra.initial_token=*token*

Use when Cassandra is not using virtual nodes (vnodes). Sets the initial partitioner token for a node the first time the node is started. (Default: disabled)

Note: Vnodes automatically select tokens.

cassandra.join_ring=*true|false*

When set to `false`, prevents the Cassandra node from joining a ring on startup. (Default: `true`) You can add the node to the ring afterwards using nodetool join and a JMX call.

cassandra.load_ring_state=*true|false*

When set to `false`, clears all gossip state for the node on restart. (Default: true)

cassandra.metricsReporterConfigFile=*file*

Enables pluggable metrics reporter. See Pluggable metrics reporting in Cassandra 2.0.2.

cassandra.native_transport_port=*port*

Sets the port on which the CQL native transport listens for clients. (Default: 9042)

cassandra.partitioner=*partitioner*

Sets the partitioner. (Default: `org.apache.cassandra.dht.Murmur3Partitioner`)

cassandra.replace_address=*listen_address* or *broadcast_address* of dead node

To replace a node that has died, restart a new node in its place specifying the listen_address or broadcast_address that the new node is assuming. The new node must be in the same state as before bootstrapping, without any data in its data directory.

Note: The *broadcast_address* defaults to the *listen_address* except when the ring is using the Ec2MultiRegionSnitch.

cassandra.replayList=*table*

Allows restoring specific tables from an archived commit log.

cassandra.ring_delay_ms=*ms*

Defines the amount of time a node waits to hear from other nodes before formally joining the ring. (Default: 1000ms)

cassandra.rpc_port=*port*

Sets the port for the Thrift RPC service, which is used for client connections. (Default: 9160).

cassandra.ssl_storage_port=*port*

Sets the SSL port for encrypted communication. (Default: 7001)

cassandra.start_native_transport=*true* | *false*

Enables or disables the native transport server. See start_native_transport in `cassandra.yaml`. (Default: true)

cassandra.start_rpc=*true* | *false*

Enables or disables the Thrift RPC server. (Default: true)

cassandra.storage_port=*port*

Sets the port for inter-node communication. (Default: 7000)

cassandra.triggers_dir=*directory*

Sets the default location for the triggers JARs.

cassandra.write_survey=*true*

Enables a tool sor testing new compaction and compression strategies. `write_survey` allows you to experiment with different strategies and benchmark write performance differences without affecting the production workload. See Testing compaction and compression on page 180.

consistent.rangemovement=*true*

Set to `true`, makes bootstrapping behavior effective.

Example

Clearing gossip state when starting a node:

- Command line: `$ bin/cassandra -Dcassandra.load_ring_state=false`
- jvm.options: `JVM_OPTS="$JVM_OPTS -Dcassandra.load_ring_state=false"`

Example

Starting a Cassandra node without joining the ring:

- Command line: `bin/cassandra -Dcassandra.join_ring=false`
- jvm.options: `JVM_OPTS="$JVM_OPTS -Dcassandra.join_ring=false"`

Example

Replacing a dead node:

- Command line: `bin/cassandra -Dcassandra.replace_address=10.91.176.160`
- jvm.options: `JVM_OPTS="$JVM_OPTS -Dcassandra.replace_address=10.91.176.160"`

The cassandra-stress tool

The `cassandra-stress` tool is a Java-based stress testing utility for basic benchmarking and load testing a Cassandra cluster.

Data modeling choices can greatly affect application performance. Significant load testing over several trials is the best method for discovering issues with a particular data model. The `cassandra-stress` tool is an effective tool for populating a cluster and stress testing CQL tables and queries. Use `cassandra-stress` to:

- Quickly determine how a schema performs.
- Understand how your database scales.
- Optimize your data model and settings.
- Determine production capacity.

The `cassandra-stress` tool also supports a YAML-based profile for defining specific schemas with various compaction strategies, cache settings, and types. Sample files are located in :

- Package installations: `/usr/share/docs/cassandra/examples`
- Tarball installations: `install_location/tools/`

The YAML file supports user-defined keyspace, tables, and schema. The YAML file can be used to design tests of reads, writes, and mixed workloads.

When started without a YAML file, `cassandra-stress` creates a keyspace, `keyspace1`, and tables, `standard1` or `counter1`, depending on what type of table is being tested. These elements are automatically created the first time you run a stress test and reused on subsequent runs. You can drop `keyspace1` using DROP KEYSPACE. You cannot change the default keyspace and tables names without using a YAML file.

Usage:

- Package installations:

  ```
  $ cassandra-stress command [options]
  ```
- Tarball installations:

  ```
  $ cd install_location/tools
  $ bin/cassandra-stress command [options]
  ```

cassandra-stress options

Command	Description
`counter_read`	Multiple concurrent reads of counters. The cluster must first be populated by a counter_write test.
`counter_write`	Multiple concurrent updates of counters.
`help`	Display help: `cassandra-stress help` Display help for an option: `cassandra-stress help [options]` For example: `cassandra-stress help -schema`
`legacy`	Legacy support mode.
`mixed`	Interleave basic commands with configurable ratio and distribution. The cluster must first be populated by a write test.
`print`	Inspect the output of a distribution definition.
`read`	Multiple concurrent reads. The cluster must first be populated by a write test.
`user`	Interleave user provided queries with configurable ratio and distribution.
`write`	Multiple concurrent writes against the cluster.

Important: Additional sub-options are available for each option in the following table. To get more detailed information on any of these, enter:

```
$ cassandra-stress help option
```

When entering the `help` command, be sure to precede the option name with a hyphen, as shown.

Cassandra-stress sub-options

Sub-option	Description

-col	Column details, such as size and count distribution, data generator, names, and comparator.
	Usage:
	`-col names=? [slice] [super=?] [comparator=?] [timestamp=?] [size=DIST(?)]`
	`or`
	`-col [n=DIST(?)] [slice] [super=?] [comparator=?] [timestamp=?] [size=DIST(?)]`

-errors	How to handle errors when encountered during stress testing. **Usage**: `-errors [retries=?] [ignore]`
-graph	Graph results of cassandra-stress tests. Multiple tests can be graphed together. **Usage**: `-graph file=? [revision=?] [title=?] [op=?]`
-insert	Insert specific options relating to various methods for batching and splitting partition updates. **Usage**: `-insert [revisit=DIST(?)] [visits=DIST(?)] partitions=DIST(?) [batchtype=?] select-ratio=DIST(?) row-population-ratio=DIST(?)`
-log	Where to log progress and the interval to use. **Usage**: `-log [level=?] [no-summary] [file=?] [interval=?]`

-mode	Thrift or CQL with options.
	Usage:
	`-mode thrift [smart] [user=?] [password=?]`
	`or`
	`-mode native [unprepared] cql3 [compression=?] [port=?] [user=?] [password=?] [auth-provider=?] [maxPending=?] [connectionsPerHost=?]`
	`or`
	`-mode simplenative [prepared] cql3 [port=?]`

-node	Nodes to connect to. **Usage**: `-node [whitelist] [file=?]`
-pop	Population distribution and intra-partition visit order.

Sub-option	Description

	Usage:
	`-pop seq=? [no-wrap] [read-lookback=DIST(?)] [contents=?]`
	`  or`
	`-pop [dist=DIST(?)] [contents=?]`

-port	Specify port for connecting Cassandra nodes. Port can be specified for Cassandra native protocol, Thrift protocol or a JMX port for retrieving statistics. **Usage**: `-port [native=?] [thrift=?] [jmx=?]`

-rate	Thread count, rate limit, or automatic mode (default is auto).
	Usage:
	`-rate threads=? [limit=?]`
	`  or`
	`-rate [threads>=?] [threads<=?] [auto]`

-sample	Specify the number of samples to collect for measuring latency. **Usage**: `-sample [history=?] [live=?] [report=?]`
-schema	Replication settings, compression, compaction, and so on. **Usage**: `-schema [replication(?)] [keyspace=?] [compaction(?)] [compression=?]`
-sendto	Specify a stress server to send this command to. **Usage**: `-sendToDaemon <host>`
-transport	Custom transport factories. **Usage**: `-transport [factory=?] [truststore=?] [truststore-password=?] [ssl-protocol=?] [ssl-alg=?] [store-type=?] [ssl-ciphers=?]`

Additional command-line parameters can modify how cassandra-stress runs:

Additional cassandra-stress parameters

Command	Description
cl=?	Set the consistency level to use during `cassandra-stress`. Options are ONE, QUORUM, LOCAL_QUORUM, EACH_QUORUM, ALL, and ANY. Default is LOCAL_ONE.
clustering=DIST(?)	Distribution clustering runs of operations of the same kind.
duration=?	Specify the time to run, in seconds, minutes or hours.

Command	Description
err<?	Specify a standard error of the mean; when this value is reached, cassandra-stress will end. Default is 0.02.
n>?	Specify a minimum number of iterations to run before accepting uncertainly convergence.
n<?	Specify a maximum number of iterations to run before accepting uncertainly convergence.
n=?	Specify the number of operations to run.
no-warmup	Do not warmup the process, do a cold start.
ops(?)	Specify what operations to run and the number of each. (only with the user option)
profile=?	Designate the YAML file to use with cassandra-stress. (only with the user option)
truncate=?	Truncate the table created during cassandra-stress. Options are never, once, or always. Default is never.

Simple read and write examples

```
# Insert (write) one million rows
$ cassandra-stress write n=1000000 -rate threads=50

# Read two hundred thousand rows.
$ cassandra-stress read n=200000 -rate threads=50

# Read rows for a duration of 3 minutes.
$ cassandra-stress read duration=3m -rate threads=50

# Read 200,000 rows without a warmup of 50,000 rows first.
$ cassandra-stress read n=200000 no-warmup -rate threads=50
```

View schema help

```
$ cassandra-stress help -schema
```

```
 replication([strategy=?][factor=?][<option 1..N>=?]):                Define
  the replication strategy and any parameters
     strategy=? (default=org.apache.cassandra.locator.SimpleStrategy)  The
  replication strategy to use
     factor=? (default=1)                                              The
  number of replicas
 keyspace=? (default=keyspace1)                                        The
  keyspace name to use
 compaction([strategy=?][<option 1..N>=?]):                            Define
  the compaction strategy and any parameters
     strategy=?                                                        The
  compaction strategy to use
 compression=?
  Specify the compression to use for SSTable, default:no compression
```

Populate the database

Generally it is easier to let cassandra-stress create the basic schema and then modify it in CQL:

```
#Load one row with default schema
$ cassandra-stress write n=1 cl=one -mode native cql3 -log
 file=create_schema.log
```

```
#Modify schema in CQL
$ cqlsh

#Run a real write workload
$ cassandra-stress write n=1000000 cl=one -mode native cql3 -schema
 keyspace="keyspace1" -log file=load_1M_rows.log
```

Change the replication strategy

Changes the replication strategy to `NetworkTopologyStrategy` and targets one node named `existing`.

```
$ cassandra-stress write n=500000 no-warmup -node existing -schema
 "replication(strategy=NetworkTopologyStrategy, existing=2)"
```

Run a mixed workload

When running a mixed workload, you must escape parentheses, greater-than and less-than signs, and other such things. This example invokes a workload that is one-quarter writes and three-quarters reads.

```
$ cassandra-stress mixed ratio\(write=1,read=3\) n=100000 cl=ONE -pop
 dist=UNIFORM\(1..1000000\) -schema keyspace="keyspace1" -mode native cql3 -
rate threads\>=16 threads\<=256 -log file=~/mixed_autorate_50r50w_1M.log
```

Notice the following in this example:

1. The `ratio` parameter requires backslash-escaped parenthesis.
2. The value of `n` used in the read phase is different from the value used in write phase. During the write phase, `n` records are written. However in the read phase, if `n` is too large, it is inconvenient to read *all* the records for simple testing. Generally, `n` does not need be large when validating the persistent storage systems of a cluster.

   The `-pop dist=UNIFORM\(1..1000000\)` portion says that of the n=100,000 operations, select the keys uniformly distributed between 1 and 1,000,000. Use this when you want to specify more data per node than what fits in DRAM.
3. In the `rate` section, the greater-than and less-than signs are escaped. If not escaped, the shell attempts to use them for IO redirection: the shell tries to read from a non-existent file called `=256` and create a file called `=16`. The `rate` section tells cassandra-stress to automatically attempt different numbers of client threads and not test less that 16 or more than 256 client threads.

Standard mixed read/write workload keyspace for a single node

```
CREATE KEYSPACE "keyspace1" WITH replication = {
  'class': 'SimpleStrategy',
  'replication_factor': '1'
};
USE "keyspace1";
CREATE TABLE "standard1" (
  key blob,
  "C0" blob,
  "C1" blob,
  "C2" blob,
  "C3" blob,
  "C4" blob,
  PRIMARY KEY (key)
) WITH
```

```
   bloom_filter_fp_chance=0.010000 AND
   caching='KEYS_ONLY' AND
   comment='' AND
   dclocal_read_repair_chance=0.000000 AND
   gc_grace_seconds=864000 AND
   index_interval=128 AND
   read_repair_chance=0.100000 AND
   replicate_on_write='true' AND
   default_time_to_live=0 AND
   speculative_retry='99.0PERCENTILE' AND
   memtable_flush_period_in_ms=0 AND
   compaction={'class': 'SizeTieredCompactionStrategy'} AND
   compression={'class': 'LZ4Compressor'};
```

Split up a load over multiple cassandra-stress instances on different nodes

This example demonstrates loading into large clusters, where a single `cassandra-stress` load generator node cannot saturate the cluster. In this example, `$NODES` is a variable whose value is a comma delimited list of IP addresses such as 10.0.0.1, 10.0.0.2, and so on.

```
#On Node1
$ cassandra-stress write n=1000000 cl=one -mode native cql3 -schema
 keyspace="keyspace1" -pop seq=1..1000000 -log file=~/node1_load.log -node
 $NODES

#On Node2
$ cassandra-stress write n=1000000 cl=one -mode native cql3 -schema
 keyspace="keyspace1" -pop seq=1000001..2000000 -log file=~/node2_load.log -
node $NODES
```

Run cassandra-stress with authentication

The following example shows using the -mode option to supply a username and password:

```
$ cassandra-stress -mode native cql3 user=cassandra password=cassandra no-
warmup cl=QUORUM
```

Note: Check the documentation of the transport option for SSL authentication.

Run cassandra-stress with authentication and SSL encryption

The following example shows using the -mode option to supply a username and password, and the -transportation option for SSL parameters:

```
$ cassandra-stress write n=100k cl=ONE no-warmup -mode native cql3
 user=cassandra password=cassandra
-transport truststore=/usr/local/lib/dsc-cassandra/conf/server-truststore.jks
 truststore-password=truststorePass
factory=org.apache.cassandra.thrift.SSLTransportFactory
keystore=/usr/local/lib/dsc-cassandra/conf/server-keystore.jks keystore-
password=myKeyPass
```

Note: Cassandra authentication and SSL encryption must already be configured before executing `cassandra-stress` with these options. The example shown above uses self-signed CA certificates.

Run cassandra-stress using the truncate option

This option must be inserted before the mode option, otherwise the cassandra-stress tool won't apply truncation as specified.

The following example shows the `truncate` command:

```
$cassandra-stress write n=100000000 cl=QUORUM truncate=always -schema
 keyspace=keyspace-rate threads=200 -log file=write_$NOW.log
```

Use a YAML file to run cassandra-stress

This example uses a YAML file named `cqlstress-example.yaml`, which contains the keyspace and table definitions, and a query definition. The keyspace name and definition are the first entries in the YAML file:

```
keyspace: perftesting

keyspace_definition:

  CREATE KEYSPACE perftesting WITH replication = { 'class':
 'SimpleStrategy', 'replication_factor': 3};
```

The table name and definition are created in the next section using CQL:

```

table: users

table_definition:

  CREATE TABLE users (
    username text,
    first_name text,
    last_name text,
    password text,
    email text,
    last_access timeuuid,
    PRIMARY KEY(username)
  );
```

In the `extra_definitions` section you can add secondary indexes or materialized views to the table:

```


extra_definitions:
  - CREATE MATERIALIZED VIEW perftesting.users_by_first_name AS SELECT *
 FROM perftesting.users WHERE first_name IS NOT NULL and username IS NOT
 NULL PRIMARY KEY (first_name, username);
  - CREATE MATERIALIZED VIEW perftesting.users_by_first_name2 AS SELECT *
 FROM perftesting.users WHERE first_name IS NOT NULL and username IS NOT
 NULL PRIMARY KEY (first_name, username);
  - CREATE MATERIALIZED VIEW perftesting.users_by_first_name3 AS SELECT *
 FROM perftesting.users WHERE first_name IS NOT NULL and username IS NOT
 NULL PRIMARY KEY (first_name, username);
```

The population distribution can be defined for any column in the table. This section specifies a uniform distribution between 10 and 30 characters for `username` values in gnerated rows, that the values in the generated rows willcreates , a uniform distribution between 20 and 40 characters for generated `startdate` over the entire Cassandra cluster, and a Gaussian distribution between 100 and 500 characters for `description` values.

```
columnspec:
  - name: username
    size: uniform(10..30)
  - name: first_name
    size: fixed(16)
  - name: last_name
    size: uniform(1..32)
  - name: password
    size: fixed(80) # sha-512
  - name: email
    size: uniform(16..50)
  - name: startdate
    cluster: uniform(20...40)
  - name: description
    size: gaussian(100...500)
```

After the column specifications, you can add specifications for how each batch runs. In the following code, the `partitions` value directs the test to use the column definitions above to insert a fixed number of rows in the partition in each batch:

```
insert:
  partitions: fixed(10)
  batchtype: UNLOGGED
```

The last section contains a query, `read1`, that can be run against the defined table.

```
queries:
  read1:
    cql: select * from users where username = ? and startdate = ?
    fields: samerow     # samerow or multirow (select arguments from the
 same row, or randomly from all rows in the partition)
```

The following example shows using the user option and its parameters to run `cassandra-stress` tests from `cqlstress-example.yaml`:

```
$ cassandra-stress user profile=tools/cqlstress-example.yaml n=1000000 ops
\(insert=3,read1=1\) no-warmup cl=QUORUM
```

Notice that:

- The user option is required for the `profile` and `opt` parameters.
- The value for the profile parameter is the path and filename of the .yaml file.
- In this example, `-n` specifies the number of *batches* that run.
- The values supplied for `ops` specifies which operations run and how many of each. These values direct the command to `insert` rows into the database and run the `read1` query.

  How many times? Each insert or query counts as one batch, and the values in `ops` determine how many of each type are run. Since the total number of batches is 1,000,000, and `ops` says to run three inserts for each query, the result will be 750,000 inserts and 250,000 of the `read1` query.

  Use escaping backslashes when specifying the `ops` value.

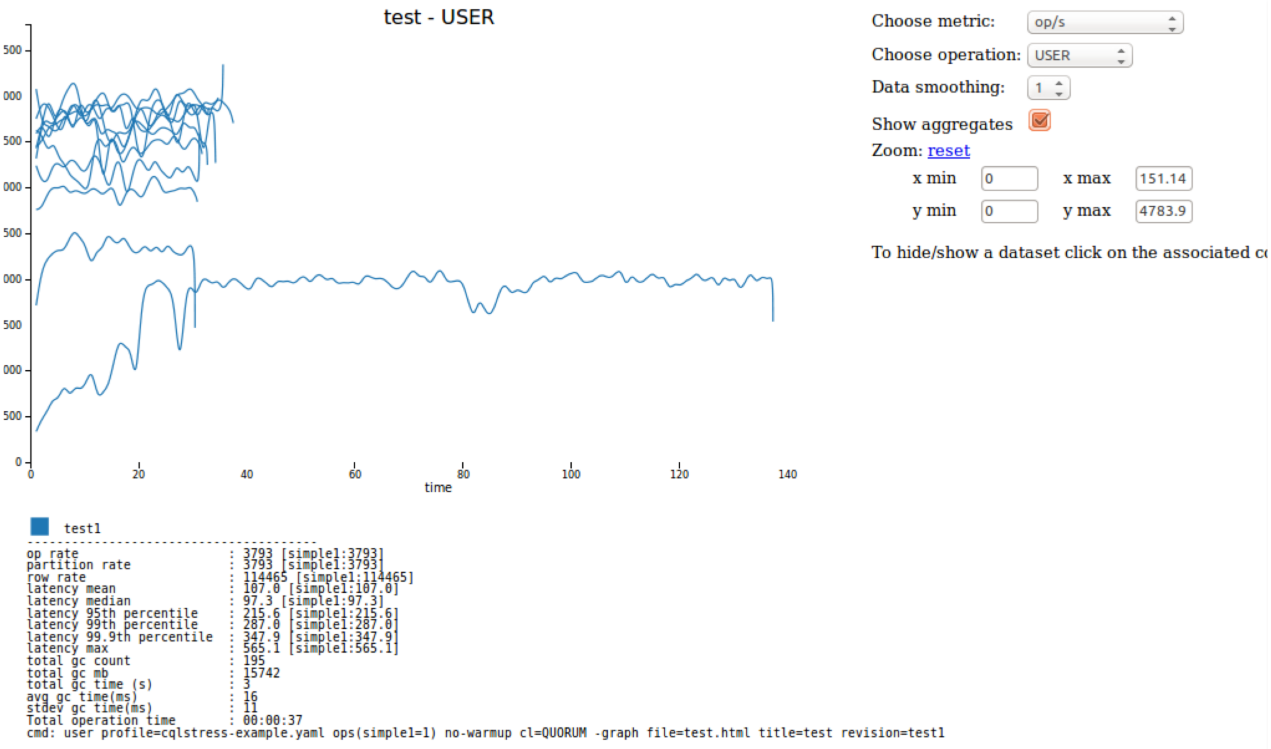
For more information, see Improved Cassandra 2.1 Stress Tool: Benchmark Any Schema – Part 1.

Use the -graph option

In Cassandra 3.2 and later, the `-graph` option provides visual feedback for `cassandra-stress` tests. A file must be named to build the resulting HTML file. A `title` and `revision` are optional, but `revision` must be used if multiple stress tests are graphed on the same output.

```
$ cassandra-stress user profile=tools/cqlstress-example.yaml ops\(insert=1\) -
graph file=test.html title=test revision=test1
```

An interactive graph can be displayed with a web browser:



Choose metric: op/s

Choose operation: USER

Data smoothing: 1

Show aggregates ☑

Zoom: reset

x min 0 x max 151.14

y min 0 y max 4783.9

To hide/show a dataset click on the associated co

🟦 test1

```
--------------------------------------
op rate                        : 3793 [simple1:3793]
partition rate                 : 3793 [simple1:3793]
row rate                       : 114465 [simple1:114465]
latency mean                   : 107.0 [simple1:107.0]
latency median                 : 97.3 [simple1:97.3]
latency 95th percentile        : 215.6 [simple1:215.6]
latency 99th percentile        : 287.0 [simple1:287.0]
latency 99.9th percentile      : 347.9 [simple1:347.9]
latency max                    : 565.1 [simple1:565.1]
total gc count                 : 195
total gc mb                    : 15742
total gc time (s)              : 3
avg gc time(ms)                : 16
stdev gc time(ms)              : 11
Total operation time           : 00:00:37
cmd: user profile=cqlstress-example.yaml ops(simple1=1) no-warmup cl=QUORUM -graph file=test.html title=test revision=test1
```

Interpreting the output of cassandra-stress

Each line reports data for the interval between the last elapsed time and current elapsed time.

```
Created keyspaces. Sleeping 1s for propagation.
   Sleeping 2s...
   Warming up WRITE with 50000 iterations...
   Running WRITE with 200 threads for 1000000 iteration
   type       total ops,    op/s,    pk/s,   row/s,    mean,     med,
  .95,     .99,    .999,     max,    time,   stderr, errors,  gc: #,  max ms,
  sum ms,  sdv ms,      mb
   total,          43148,   42991,   42991,   42991,     4.6,     1.5,
 10.9,   106.1,   239.3,   255.4,     1.0,  0.00000,      0,      1,      49,
     49,       0,     612
   total,          98715,   43857,   43857,   43857,     4.6,     1.7,
 8.5,     98.6,   204.6,   264.5,     2.3,  0.00705,      0,      1,      45,
     45,       0,     619
   total,         157777,   47283,   47283,   47283,     4.1,     1.4,
 8.3,     70.6,   251.7,   286.3,     3.5,  0.02393,      0,      1,      59,
     59,       0,     611

   Results:
   op rate                   : 46751 [WRITE:46751]
   partition rate            : 46751 [WRITE:46751]
   row rate                  : 46751 [WRITE:46751]
   latency mean              : 4.3 [WRITE:4.3]
   latency median            : 1.3 [WRITE:1.3]
   latency 95th percentile   : 7.2 [WRITE:7.2]
   latency 99th percentile   : 60.5 [WRITE:60.5]
   latency 99.9th percentile : 223.2 [WRITE:223.2]
   latency max               : 503.1 [WRITE:503.1]
   Total partitions          : 1000000 [WRITE:1000000]
   Total errors              : 0 [WRITE:0]
   total gc count            : 18
   total gc mb               : 10742
   total gc time (s)         : 1
   avg gc time(ms)           : 73
   stdev gc time(ms)         : 16
   Total operation time      : 00:00:21

   END
```

Table: Output of cassandra-stress

Data	Description
`total ops`	Running total number of operations during the run.
`op/s`	Number of operations per second performed during the run.
`pk/s`	Number of partition operations per second performed during the run.
`row/s`	Number of row operations per second performed during the run.
`mean`	Average latency in milliseconds for each operation during that run.
`med`	Median latency in milliseconds for each operation during that run.
`.95`	95% of the time the latency was less than the number displayed in the column.
`.99`	99% of the time the latency was less than the number displayed in the column.
`.999`	99.9% of the time the latency was less than the number displayed in the column.
`max`	Maximum latency in milliseconds.
`time`	Total operation time.
`stderr`	Standard error of the mean. It is a measure of confidence in the average throughput number; the smaller the number, the more accurate the measure of the cluster's performance.
`gc: #`	Number of garbage collections.
`max ms`	Longest garbage collection in milliseconds.
`sum ms`	Total of garbage collection in milliseconds.
`sdv ms`	Standard deviation in milliseconds.
`mb`	Size of the garbage collection in megabytes.

SSTable utilities

sstabledump

This tool outputs the contents of the specified SSTable in the JSON format.

Depending on your task, you may wish to flush the table to disk (using nodetool flush)before dumping its contents.

Usage:

- Package installations:

```
$ sstabledump [options]  sstable_file
```

- Tarball installations:

```
$ cd install_location
$ bin/sstabledump [options] sstable_file
```

The file is located in the `data` directory and has a `.db` extension.

Table: Options

Flag	Description
-d	Outputs an internal representation, one CQL row per line.
-e	Limits output to the list of keys.
-k *key*	Limits output to information about the row identified by the specified key.
-x*key*	Excludes information about the row identified by the specified key

Flag	Description
	from output.

sstableexpiredblockers

During compaction, Cassandra can drop entire SSTables if they contain only expired tombstones and if it is guaranteed to not cover any data in other SSTables. This diagnostic tool outputs all SSTables that are blocking other SSTables from being dropped.

Usage:

- Package installations: `$ sstableexpiredblockers [--dry-run]` *`keyspace table`*
- Tarball installations:

```
$ cd install_location/tools
$ bin/sstableexpiredblockers [--dry-run] keyspace table
```

Procedure

Choose a keyspace and table to check for any SSTables that are blocking the specified table from dropping.

```
$ sstableexpiredblockers cycling cyclist_name
```

What to do next

sstablekeys

The `sstablekeys` utility dumps table keys.

Usage:

- Package installations: `$ sstablekeys` *`sstable_name`*
- Tarball installations:

```
$ cd install_location/tools
$ bin/sstablekeys sstable_name
```

Procedure

1. If data has not been previously flushed to disk, manually flush it. For example:

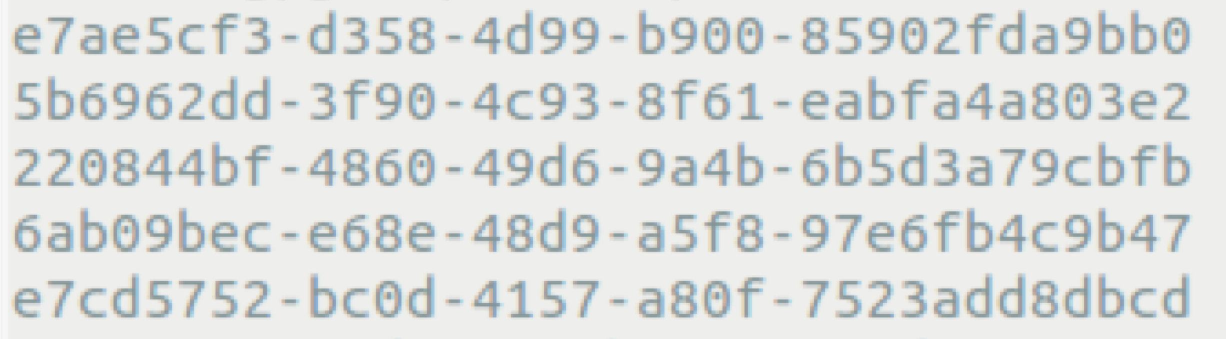
   ```
   $ nodetool flush cycling cyclist_name
   ```
2. To list the keys in an SSTable, find the name of the SSTable file.

   The file is located in the `data` directory and has a `.db` extension.
3. Look at keys in the SSTable data. For example, use `sstablekeys` followed by the path to the data. Use the path to data for your Cassandra installation:

   ```
   ## Package installations
   $ sstablekeys /var/lib/cassandra/data/cycling/cyclist_name-
   a882dca02aaf11e58c7b8b496c707234/la-1-big-Data.db
   ```

```
## Tarball installations
$ sstablekeys install_location/data/data/cycling/cyclist_name-
a882dca02aaf11e58c7b8b496c707234/la-1-big-Data.db
```

The output appears, for example:



```
e7ae5cf3-d358-4d99-b900-85902fda9bb0
5b6962dd-3f90-4c93-8f61-eabfa4a803e2
220844bf-4860-49d6-9a4b-6b5d3a79cbfb
6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47
e7cd5752-bc0d-4157-a80f-7523add8dbcd
```

sstablelevelreset

Reset level to 0 on a given set of SSTables that use LeveledCompactionStrategy.

Usage:

- Package installations: `$ sstablelevelreset [--really-reset] keyspace table`
- Tarball installations:

```
$ cd install_location/tools
$ bin/sstablelevelreset [--really-reset] keyspace table
```

The option `--really-reset` is a warning that Cassandra is stopped before the tool is run.

Procedure

- Stop Cassandra on the node. Choose a keyspace and table to reset to level 0.

```
$ sstablelevelreset --really-reset cycling cyclist_name
```

  If the designated table is already at level 0, then no change occurs. If the SSTable is releveled, the metadata is rewritten to designate the level to 0.

Example

sstableloader (Cassandra bulk loader)

The Cassandra bulk loader, also called the sstableloader, provides the ability to:

- Bulk load external data into a cluster.
- Load existing SSTables into another cluster with a different number of nodes or replication strategy.
- Restore snapshots.

The `sstableloader` streams a set of SSTable data files to a live cluster. It does not simply copy the set of SSTables to every node, but transfers the relevant part of the data to each node, conforming to the replication strategy of the cluster. The table into which the data is loaded does not need to be empty.

Run `sstableloader` from the directory containing the SSTables, passing it the location of the target cluster.

Note: Bulkloading SSTables created in versions prior to Cassandra 3.0 is supported only in Cassandra 3.0.5 and later.

Note: Repairing tables that have been loaded into a different cluster does not repair the source tables.

Prerequisites

The source data loaded by `sstableloader` must be in SSTables.

Generating SSTables

When using `sstableloader` to load external data, you must first put the external data into SSTables.

Note: If using DataStax Enterprise, you can use Sqoop to migrate external data to Cassandra.

SSTableWriter is the API to create raw Cassandra data files locally for bulk load into your cluster. The Cassandra source code includes the CQLSSTableWriter implementation for creating SSTable files from external data without needing to understand the details of how those map to the underlying storage engine. Import the `org.apache.cassandra.io.sstable.CQLSSTableWriter` class, and define the schema for the data you want to import, a writer for the schema, and a prepared insert statement. For a complete example, see http://www.datastax.com/dev/blog/using-the-cassandra-bulk-loader-updated.

Restoring Cassandra snapshots

For information about preparing snapshots for `sstableloader` import, see Restoring from centralized backups.

Importing SSTables from an existing cluster

Before importing existing SSTables, run nodetool flush on each source node to assure that any data in memtables is written out to the SSTables on disk.

Preparing the target environment

Before loading the data, you must define the schema of the target tables with CQL or Thrift.

Usage

Package installations:

```
$ sstableloader -d host_url (,host_url …) [options] path_to_keyspace
```

Tarball installations:

```
$ cd install_location/bin
$ sstableloader -d host_url (,host_url …) [options] path_to_keyspace
```

The `sstableloader` bulk loads the SSTables found in the keyspace directory to the configured target cluster, where the parent directories of the directory path are used as the target keyspace/table name.

For more `sstableloader` options, see sstableloader options

Using sstableloader

1. If restoring snapshot data from some other source: make sure that the snapshot files are in a `keyspace/tablename` directory path whose names match those of the target `keyspace/tablename`. In this example, make sure the snapshot files are in `/Keyspace/Standard1/`.
2. Go to the location of the SSTables:

   Package installations:

   ```
   $ cd /var/lib/cassandra/data/Keyspace1/Standard1/
   ```

   Tarball installations:

```
$ cd install_location/data/data/Keyspace1/Standard1/
```

3. To view the contents of the keyspace:

   ```
   $ ls

   Keyspace1-Standard1-jb-60-CRC.db
   Keyspace1-Standard1-jb-60-Data.db
   ...
   Keyspace1-Standard1-jb-60-TOC.txt
   ```

4. To bulk load the files, specify the path to `Keyspace1/Standard1/` in the target cluster:

   ```
   $ sstableloader -d 110.82.155.1 /var/lib/cassandra/data/Keyspace1/Standard1/
    ##  Package installation

   $ install_location/bin/sstableloader -d 110.82.155.1 /var/lib/cassandra/
   data/data/Keyspace1/Standard1/ ## Tarball installation
   ```

   This command bulk loads all files.

Note: To get the best throughput from SSTable loading, you can use multiple instances of sstableloader to stream across multiple machines. No hard limit exists on the number of SSTables that sstableloader can run at the same time, so you can add additional loaders until you see no further improvement.

Table: sstableloader options

Short option	Long option	Description
-alg	--ssl-alg <ALGORITHM>	Client SSL algorithm (default: SunX509).
-ap	--auth-provider <auth provider class name>	Allows the use of a third party auth provider. Can be combined with -u <username> and -pw <password> if the auth provider supports plain text credentials.
-ciphers	--ssl-ciphers <CIPHER-SUITES>	Client SSL. Comma-separated list of encryption suites.
-cph	--connections-per-host <connectionsPerHost>	Number of concurrent connections-per-host.
-d	--nodes <initial_hosts>	**Required**. Connect to a list of (comma separated) hosts for initial cluster information.
-f	--conf-path <path_to_config_file>	Path to the `cassandra.yaml` path for streaming throughput and client/server SSL.
-h	--help	Display help.
-i	--ignore <NODES>	Do not stream to this comma separated list of nodes.
-ks	--keystore <KEYSTORE>	Client SSL. Full path to the keystore.
-kspw	--keystore-password <KEYSTORE-PASSWORD>	Client SSL. Password for the keystore. Overrides the client_encryption_options option in cassandra.yaml
	--no-progress	Do not display progress.
-p	--port <rpc port>	RPC port (default: 9160 [Thrift]).
-prtcl	--ssl-protocol <PROTOCOL>	Client SSL. Connections protocol to use (default: TLS).

Short option	Long option	Description
		Overrides the server_encryption_options option in cassandra.yaml
-pw	--password <password>	Password for Cassandra authentication.
-st	--store-type <STORE-TYPE>	Client SSL. Type of store.
-t	--throttle <throttle>	Throttle speed in Mbits (default: unlimited). Overrides the stream_throughput_outbound_megabits_per_sec option in cassandra.yaml
-tf	--transport-factory <transport factory>	Fully-qualified `ITransportFactory` class name for creating a connection to Cassandra.
-ts	--truststore <TRUSTSTORE>	Client SSL. Full path to truststore.
-tspw	--truststore-password <TRUSTSTORE-PASSWORD>	Client SSL. Password of the truststore.
-u	--username <username>	User name for Cassandra authentication.
-v	--verbose	Verbose output.

sstablemetadata

The `sstablemetadata` utility prints metadata about a specified SSTable, including:

- SSTable name
- partitioner
- SSTable level (for Leveled Compaction only)
- number of tombstones and Dropped timestamps (in epoch time)
- number of cells and size (in bytes) per row

Use this report to troubleshoot wide rows or performance-degrading tombstones.

Procedure

1. Switch to the `CASSANDRA_HOME` directory.
2. Enter the command `/tools/bin/sstablemetadata` followed by the filenames of one or more SSTables.

```
$ tools/bin/sstablemetadata <sstable_name filenames>
```

```
tools/bin/sstablemetadata  data/data/autogeneratedtest/
transaction_by_retailer-f27e4d5078dc11e59d629d03f52e8a2b/ma-203-big-Data.db
SSTable: data/data/autogeneratedtest/transaction_by_retailer-
f27e4d5078dc11e59d629d03f52e8a2b/ma-203-big
Partitioner: org.apache.cassandra.dht.Murmur3Partitioner
Bloom Filter FP chance: 0.010000
Minimum timestamp: 1445871871053006
Maximum timestamp: 1445871953354005
SSTable max local deletion time: 2147483647
```

```
Compression ratio: -1.0
Estimated droppable tombstones: 0.0
SSTable Level: 0
Repaired at: 0
ReplayPosition(segmentId=1445871179392, position=18397674)
Estimated tombstone drop times:
2147483647:   7816721
Count               Row Size        Cell Count
1                          0                 0
2                          0                 0
3                          0                 0
4                          0                 0
5                          0                 0
6                          0                 0
7                          0                 0
8                          0                 0
10                         0                 0
12                         0            710611
14                         0                 0
17                         0                 0
20                         0                 0
24                         0                 0
29                         0                 0
35                         0                 0
42                         0                 0
50                         0                 0
60                         0                 0
72                         0                 0
86                         0                 0
103                        0                 0
124                        0                 0
149                        0                 0
179                        0                 0
215                        0                 0
258                        0                 0
310                       81                 0
372                   710530                 0
446                        0                 0
535                        0                 0
642                        0                 0
770                        0                 0
924                        0                 0
1109                       0                 0
1331                       0                 0
1597                       0                 0
1916                       0                 0
2299                       0                 0
2759                       0                 0
3311                       0                 0
3973                       0                 0
4768                       0                 0
5722                       0                 0
6866                       0                 0
8239                       0                 0
9887                       0                 0
11864                      0                 0
14237                      0                 0
17084                      0                 0
20501                      0                 0
24601                      0                 0
29521                      0                 0
35425                      0                 0
42510                      0                 0
51012                      0                 0
```

```
61214                       0                 0
73457                       0                 0
88148                       0                 0
105778                      0                 0
126934                      0                 0
152321                      0                 0
182785                      0                 0
219342                      0                 0
263210                      0                 0
315852                      0                 0
379022                      0                 0
454826                      0                 0
545791                      0                 0
654949                      0                 0
785939                      0                 0
943127                      0                 0
1131752                     0                 0
1358102                     0                 0
1629722                     0                 0
1955666                     0                 0
2346799                     0                 0
2816159                     0                 0
3379391                     0                 0
4055269                     0                 0
4866323                     0                 0
5839588                     0                 0
7007506                     0                 0
8409007                     0                 0
10090808                    0                 0
12108970                    0                 0
14530764                    0                 0
17436917                    0                 0
20924300                    0                 0
25109160                    0                 0
30130992                    0                 0
36157190                    0                 0
43388628                    0                 0
52066354                    0                 0
62479625                    0                 0
74975550                    0                 0
89970660                    0                 0
107964792                   0                 0
129557750                   0                 0
155469300                   0                 0
186563160                   0                 0
223875792                   0                 0
268650950                   0                 0
322381140                   0                 0
386857368                   0                 0
464228842                   0                 0
557074610                   0                 0
668489532                   0                 0
802187438                   0                 0
962624926                   0                 0
1155149911                  0                 0
1386179893                  0                 0
1663415872                  0                 0
1996099046                  0                 0
2395318855                  0                 0
2874382626                  0
3449259151                  0
4139110981                  0
4966933177                  0
5960319812                  0
```

```
7152383774                 0
8582860529                 0
10299432635                 0
12359319162                 0
14831182994                 0
17797419593                 0
21356903512                 0
25628284214                 0
30753941057                 0
36904729268                 0
44285675122                 0
53142810146                 0
63771372175                 0
76525646610                 0
91830775932                 0
110196931118                 0
132236317342                 0
158683580810                 0
190420296972                 0
228504356366                 0
274205227639                 0
329046273167                 0
394855527800                 0
473826633360                 0
568591960032                 0
682310352038                 0
818772422446                 0
982526906935                 0
1179032288322                 0
1414838745986                 0
Estimated cardinality: 722835
```

sstableofflinerelevel

This tool is intended to run in an offline fashion. When using the `LevelledCompactionStrategy`, it is possible for the number of SSTables in level L0 to become excessively large, resulting in read latency degrading. This is often the case when atypical write load is experienced (eg. bulk import of data, node bootstrapping). This tool will relevel the SSTables in an optimal fashion. The `--dry run` flag can be used to run in test mode and examine the tools results.

Usage:

- Package installations: `$ sstableofflinerelevel [--dry-run] keyspace table`
- Tarball installations:

```
$ cd install_location/tools
$ bin/sstableofflinerelevel [--dry-run] keyspace table
```

Procedure

Choose a keyspace and table to relevel.

```
$ sstableofflinerelevel cycling cyclist_name
```

sstablerepairedset

This tool is intended to mark specific SSTables as repaired or unrepaired. It is used to set the `repairedAt` status on a given set of SSTables. This metadata facilitates incremental repairs. It can take in the path to an individual SSTable or the path to a file containing a list of SSTables paths.

Warning: Do not run this command until you have stopped Cassandra on the node.

Use this tool in the process of migrating a Cassandra installation to incremental repair.

Usage:

- Package installations:

  ```
  $ sstablerepairedset [--really-set] [--is-repaired | --is-unrepaired] [-
  f sstable-list | sstables]
  ```
- Tarball installations:

  ```
  $ cd install_location/tools
  $ bin/sstablerepairedset [--really-set] [--is-repaired | --is-unrepaired] [-
  f sstable-list | sstables]
  ```

Procedure

-
- Choose SSTables to mark as repaired.

  ```
  $ sstablerepairedset --really-set --is-repaired data/data/cycling/
  cyclist_name-a882dca02aaf11e58c7b8b496c707234/la-1-big-Data.db
  ```
- Use a file to list the SSTable to mark as unrepaired.

  ```
  $ /sstablerepairedset --is-unrepaired -f repairSetSSTables.txt
  ```

  A file like `repairSetSSTables.txt` would contain a list of SSTable (`.db`) files, as in the following example:

  ```
  /data/data/cycling/cyclist_by_country-82246fc065ff11e5a4c58b496c707234/ma-1-
  big-Data.db
  /data/data/cycling/cyclist_by_birthday-8248246065ff11e5a4c58b496c707234/
  ma-1-big-Data.db
  /data/data/cycling/cyclist_by_birthday-8248246065ff11e5a4c58b496c707234/
  ma-2-big-Data.db
  /data/data/cycling/cyclist_by_age-8201305065ff11e5a4c58b496c707234/ma-1-big-
  Data.db
  /data/data/cycling/cyclist_by_age-8201305065ff11e5a4c58b496c707234/ma-2-big-
  Data.db
  ```

  Use the following command to list all the `Data.db` files in a keyspace:

  ```
  find '/home/user/datastax-ddc-3.2.0/data/data/keyspace1/' -iname "*Data.db*"
  ```

sstablescrub

The sstablescrub utility is an offline version of nodetool scrub. It attempts to remove the corrupted parts while preserving non-corrupted data. Because sstablescrub runs offline, it can correct errors that `nodetool scrub` cannot. If an SSTable cannot be read due to corruption, it will be left on disk.

If scrubbing results in dropping rows, new SSTables become unrepaired. However, if no bad rows are detected, the SSTable keeps its original `repairedAt` field, which denotes the time of the repair.

Procedure

1. Before using `sstablescrub`, try rebuilding the tables using `nodetool scrub`.

   If `nodetool scrub` does not fix the problem, use this utility.
2. Shut down the node.
3. Run the utility:

   - Package installations:

     ```
     $ sstablescrub [options] keyspace table
     ```
   - Tarball installations:

     ```
     $ cd install_location
     $ bin/sstablescrub [options] keyspace table
     ```

   Table: Options

Flag	Option	Description
	--debug	Display stack traces.
-h	--help	Display help.
-m	--manifest-check	Only check and repair the leveled manifest, without actually scrubbing the SSTables.
-s	--skip-corrupted	Skip corrupt rows in counter tables.
-v	--verbose	Verbose output.

sstablesplit

Use this tool to split SSTables files into multiple SSTables of a maximum designated size. For example, if SizeTieredCompactionStrategy was used for a major compaction and results in a excessively large SSTable, it's a good idea to split the table because won't get compacted again until the next huge compaction.

Cassandra must be stopped to use this tool:

- Package installations:

  ```
  $ sudo service cassandra stop
  ```
- Tarball installations:

  ```
  $ ps auwx | grep cassandra
  $ sudo kill pid
  ```

Usage:

- Package installations: `$ sstablesplit [options] <filename> [<filename>]*`
- Tarball installations:

  ```
  $ cd install_location/tools/bin
  sstablesplit [options] <filename> [<filename>]*
  ```

Example:

```
$ sstablesplit -s 40 /var/lib/cassandra/data/data/Keyspace1/Standard1/*
```

Table: Options

Flag	Option	Description
	--debug	Display stack traces.
-h	--help	Display help.
	--no-snapshot	Do not snapshot the SSTables before splitting.
-s	--size <size>	Maximum size in MB for the output SSTables (default: 50).
-v	--verbose	Verbose output.

sstableupgrade

This tool rewrites the SSTables in the specified table to match the currently installed version of Cassandra.

If restoring with sstableloader, you must upgrade your snapshots before restoring for any snapshot taken in a major version older than the major version that Cassandra is currently running.

Usage:

- Package installations:

```
$ sstableupgrade [options] keyspace table [snapshot]
```

- Tarball installations:

```
$ cd install_location
$ bin/sstableupgrade [options] keyspace table [snapshot]
```

The snapshot option only upgrades the specified snapshot.

Table: Options

Flag	Option	Description
	--debug	Display stack traces.
-h	--help	Display help.

sstableutil

The sstableutil will list the SSTable files for a provided table.

Usage:

- Package installations: `$ sstableutil [--cleanup | --debug | --help | --opslog | --type <arg> | --verbose] keyspace | table`
- Tarball installations:

```
$ cd install_location$ bin/sstableutil [--cleanup | --debug | --help | --
opslog | --type <arg> | --verbose] keyspace | table
```

Note: Arguments for `--type` option are: all, tmp, or final.

Procedure

Choose a table fof which to list SSTables files.

```
$ sstableutil --all cycling cyclist_name
```

sstableverify

The sstableverify utility will verify the SSTable for a provided table and look for errors or data corruption.

Usage:

- Package installations: `$ sstableverify [--debug | --extended | --help | --verbose] keyspace | table`
- Tarball installations:

```
$ cd install_location$ bin/sstableverify [--debug | --extended | --help | --verbose] keyspace | table
```

Procedure

Choose a table to verify.

```
$ sstableverify --verbose cycling cyclist_name
```

Troubleshooting

Troubleshooting has moved to Troubleshooting for both Cassandra and DataStax Enterprise.

DataStax Distribution of Apache Cassandra 3.x release notes

Note: Cassandra is now releasing on a tick-tock schedule.

The latest version of DataStax Distribution of Apache Cassandra 3.x is 3.9.

The CHANGES.txt describes the changes in detail. You can view all version changes by branch or tag in the drop-down list on the changes page.

New features, improvements, and notable changes are described in What's new?.